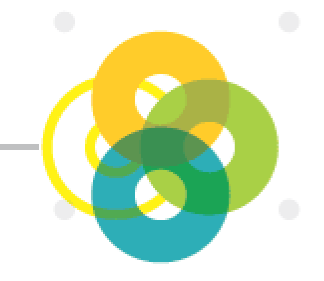
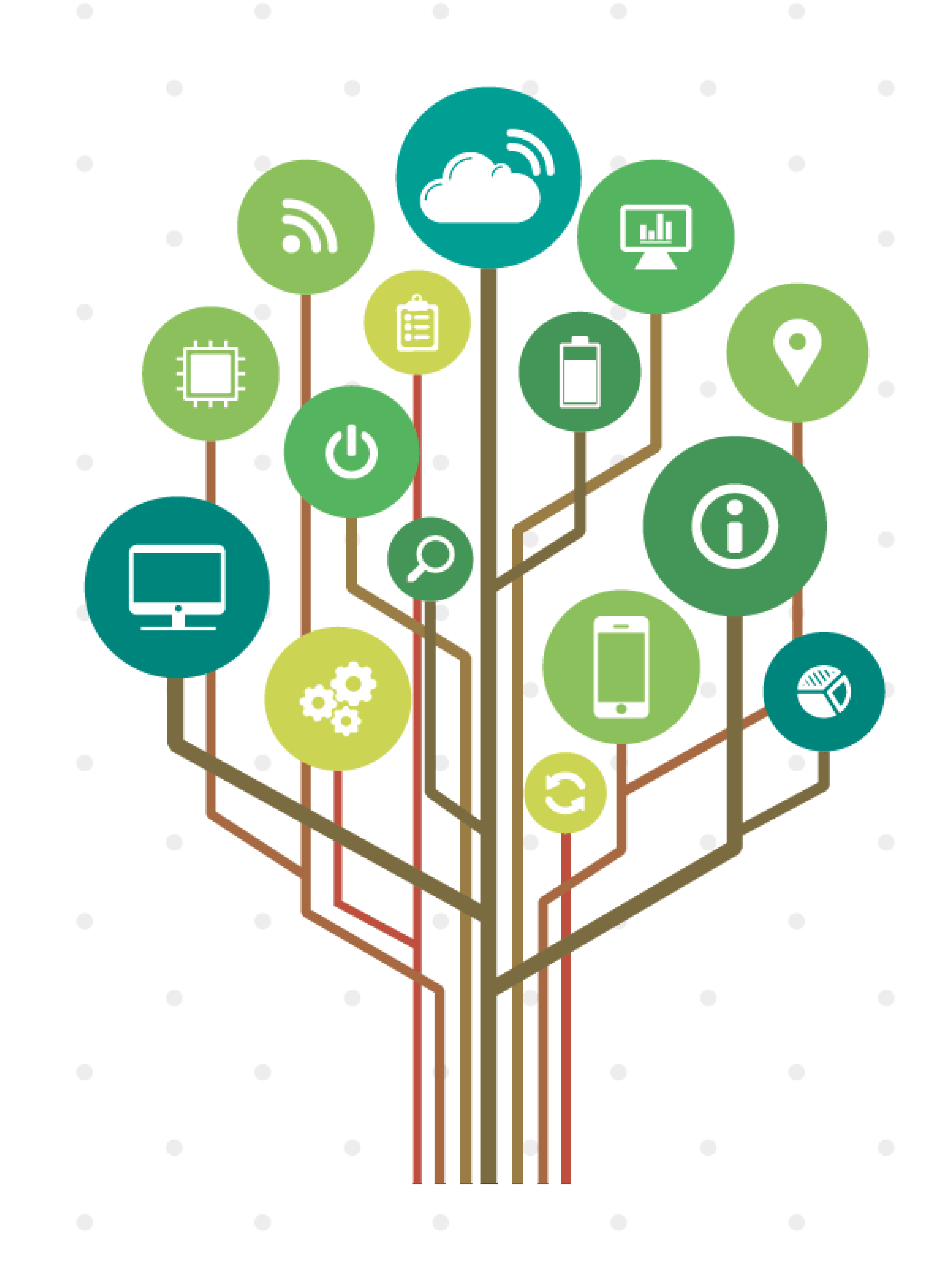


大数据技术与应用专业规划教材
教育部–阿里云产学合作专业综合改革项目规划教材

大数据基础及应用

◎ 吕云翔 钟巧灵 衣志昊 编著



清華大学出版社

大数据技术与应用专业规划教材

教育部-阿里云产学合作专业综合改革项目规划教材

大数据基础及应用

吕云翔　钟巧灵　衣志昊　编著

清华大学出版社

北京

内 容 简 介

本书从大数据的基本概念开始，由浅入深地领会大数据的精髓。本书除了讲述必要的大数据理论之外，还通过大数据实践来讲述大数据技术的应用，包括如何运用阿里云大数据计算平台分析和解决实际问题，很好地体现了大数据理论与实践的有机结合。

本书分为三大部分，分别是大数据概述及基础、大数据处理和大数据分析与应用。其中，大数据概述及基础部分重点介绍数据组织、重要数据结构、大数据协同技术以及大数据存储技术等内容；大数据处理部分重点介绍大数据处理框架，包括大数据批处理和流处理框架等内容；大数据分析与应用部分重点介绍数据分析技术和机器学习的相关内容，以及如何利用阿里云的数加平台进行基本的大数据开发工作。

本书既可以作为高等院校计算机科学、软件工程及相关专业"大数据"课程的教材，也可以供系统分析师、系统架构师、软件开发工程师和项目经理，以及其他准备或正在学习大数据技术的读者（包括参加计算机等级考试或相关专业自学考试的人员）阅读和参考。

图书在版编目（CIP）数据

大数据基础及应用/吕云翔等编著．—北京：清华大学出版社，2017
（大数据技术与应用专业规划教材）
ISBN 978-7-302-46691-8

Ⅰ．①大…　Ⅱ．①吕…　Ⅲ．数据处理　Ⅳ．①TP274

中国版本图书馆 CIP 数据核字（2017）第 038743 号

责任编辑：魏江江　王冰飞
封面设计：刘　键
责任校对：焦丽丽
责任印制：宋　林

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编**：100084
社 总 机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课件下载：http://www.tup.com.cn，010-62795954
印 装 者：清华大学印刷厂
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　　张**：15.75　　**字　　数**：311 千字
版　　次：2017 年 3 月第 1 版　　**印　　次**：2017 年 3 月第 1 次印刷
印　　数：1～2000
定　　价：39.50 元

产品编号：072901-01

序 言

PREFACE

DT 时代的数据思维与智能思维

本套云计算大数据丛书出版正值信息科技领域进入新一轮巨变，中国经济面临转型机遇的特殊时期。全球信息科技行业伴随着云计算、大数据、物联网、人工智能的发展即将进入一个泛智能的时代，云计算成为数字经济的基础设施；数据驱动、泛在智能成为各行各业转型升级的基础，不仅传统的 IT 从业人员面临能力升级，大多数在校大学生也面临新一轮知识体系的更新，各个垂直行业面临新一轮的人才升级。新一代人才教育与培训，需要一套产学一体的培训课程体系，这是阿里云愿意投身云计算大数据网络安全人才培养体系的时代背景。云计算、大数据、网络安全不仅关乎网络强国的大使命，也逐步成为各行各业专业人才的"元学科"，会逐步成为高等与职业教育的通识课程，一些发达国家已经在中小学立法普及编程课，已经开始指向这个趋势。"懂云计算，有数据思维，理解智能化"，未来可能是每一个工程技术人员与专业人士的必要素质。

2016 年开始，全球信息科技进入一个新的加速爆发周期，可能发生的大概率事件是：二十年之内，有一半的人类知识工作者会被人工智能替代，有服务能力的机器人会诞生，全世界的产业工人会少于机器人；虚拟现实和增强现实会替代今天的智能手机，变成一个新的入口；各行各业都会需要基于物联网的智能化，"中国制造"会成为广泛意义的"中国智造"。

新一轮科技带来了生活方式的变革、生产方式的变革，还有学习方式的变革，这几个趋势的背后，是云计算作为一种普惠科技的基础设施，大数据成为新能源，智能化成为一种新常识。

2016 年，全世界的短视频总量增长了 6 倍，直播业务在中国增长了 10 倍，远在偏远小镇的青年可以通过直播做电子商务，转化率可以提升十倍以上。当一个技术的使用成本趋近于零的时候，会带来广泛的社会效应。十年以前的直播只有电视台能做，需要专门的摄像机等设备，而今天的直播只需要一个手机，而且是多对多带互动的。无论是短视频，还是直播，背后都有云计算作为普惠科技的支撑作用，由此带来的，所有与知识传播有关的教育，包括整个内容行业，都会被它改变，随着大数据和人工智能的加入，人类学习的方式交互性会更强，“学习系统”会根据不同人的理解程度做个性化的推荐与辅导。

这意味着知识生产与知识传播方式的根本性转变，这个恰恰是云计算、人工智能等科技与各行各业产生化学反应的交叉点，数据是这个转变的新能源。

在 2016 年 10 月，阿里云和法院系统合作，发布了一个面向法律服务的智能应用“法小淘”，通过把数千万份法律判例文本化，“法小淘”智能应用可以为普通老百姓以及初级律师提供“打官司”的咨询服务，根据用户输入的案件信息给出建议，包括推荐合适的律师。貌似与科技远离的法律服务也用上了人工智能，这是垂直行业泛智能化的一个小例子。

中国制造进入智能时代

在工业界，阿里云跟中石化合作，协助他们做了企业的电商平台；与徐工合作，推动工厂基于工业云的智能化；与上汽合作，推出具有智能服务的互联网汽车，都收到积极的市场反馈。中国制造，面临智能化的产业机遇，借助互联网人口和产业布局两大优势成为未来的第一个智能产品制造国。

在接下来的几年，互联网+智能制造的叠加会在很多个垂直领域出现，数据智能与制造业结合，产生“跨界重混”的效果，甚至制造业就不是以制造为主，而是以服务化为主。这个巨大的重构背后依赖云和大数据。也因为这个需求，我们可预见工业企业对云计算大数据人才的需求会越来越强烈。

“创业化生存”与共享经济的兴起

创业化，会成为一种常态，越来越多的年轻人开始告别公司，兴起中的数字经济体都是基于云平台的网络化协作组织；云计算成为共享经济的超级容器，催生新一代创业者和“斜杠青年”。十年以后，或许一半以上的从业者都是“斜杠青年”，今天美国就有数千万人是跨工作、跨公司的“斜杠青年”。

过去十年，云计算使得创业公司的创业门槛降低了 10 倍，没有云计算，Airbnb、NetFlix、推特、Uber 等公司不可能这么快成长壮大，新一代创业者的一个核心能力就是要懂技术，理解数据和算法的价值，缺少技术理解力的创业者将面临更大的同质化压力。一句话，无论是草根创业，还是做一个“斜杠青年”，必要的数据思维是生存本能。

创业化和共享经济的崛起,有赖于云计算作为基础设施,大数据作为新能源的全新范式,新一代创业公司需要大量的科技人才。

在未来的经济环境里,普惠云科技的基础设施化、制造的智能化、软件的泛化以及数据无处不在,是一个大趋势,并且不断向各行各业渗透。本套丛书就是希望在这个普惠科技与各行各业深度融合的时代为下一代科技人才的培养提供更多产业界的经验与实践。

感谢清华大学出版社出版本套云计算与大数据方面的系列教材。感谢各位高校老师的辛苦努力和用心付出,使得本系列教材能够付梓出版。

——阿里云业务总经理 刘松

前言

FOREWORD

互联网技术不断发展，各种技术不断涌现，其中大数据技术已成为一颗闪耀的新星。我们已经处于数据世界，互联网每天产生大量的数据，利用好这些数据可以给我们的生活带来巨大的变化以及提供极大的便利。目前大数据技术受到越来越多的机构的重视，因为大数据技术可以给其创造巨大的利润，其中的典型代表是个性化推荐以及大数据精准营销。

本书在讲述大数据的基本概念、原理与方法的基础上，详细而全面地介绍了可以实际用于大数据实践的各种技能，旨在使学生通过有限课时的学习后，不仅能对大数据技术的基本原理有所认识，而且能够具备基本的大数据技术开发能力以及运用大数据技术解决基本的数据分析问题，理解大数据框架（尤其是阿里云大数据计算平台），在阿里云大数据平台上进行基本的大数据开发工作的能力。

本书分为三大部分，分别是大数据概述及基础、大数据处理和大数据分析与应用。其中，大数据概述及基础部分重点介绍数据组织、重要数据结构、大数据协同技术以及大数据存储技术等内容；大数据处理部分重点介绍大数据处理框架，包括大数据批处理和流处理框架等内容；大数据分析与应用部分重点介绍数据分析技术和机器学习的相关内容，以及如何利用阿里云的数加平台进行基本的大数据开发工作。

本书与其他类似著作的不同之处在于，除了讲述必要的大数据理论之外，还通过大数据实践来讲述大数据技术的应用，包括如何运用阿里云大数据计算平台解决和分析实际的问题，如阿里云 MaxCompute 和 StreamCompute 等。本书的最后一章“大数据实践：基于数加平台的推荐系统”是学生在做课程设计时可供模仿的一个项目，它完整地体现了理论与实践的有机结合。

本书的理论知识的教学安排建议如下。

章　节	内　　容	学　时　数
第 1 章	大数据概念和发展背景	1
第 2 章	大数据系统架构概述	1～2
第 3 章	分布式通信与协同	2～4
第 4 章	大数据存储	4～6
第 5 章	分布式处理	2
第 6 章	Hadoop MapReduce 解析	2～4
第 7 章	Spark 解析	2～4
第 8 章	流计算	2
第 9 章	图计算	2
第 10 章	阿里云大数据计算服务平台	2
第 11 章	集群资源管理与调度	4～6
第 12 章	数据分析	2～4
第 13 章	数据挖掘与机器学习技术	2～4
第 14 章	大数据实践：基于数加平台的推荐系统	4～5

建议理论教学时数：32～48 学时。

建议实验(实践)教学时数：16～32 学时。

教师可以按照自己对大数据的理解适当地删除一些章节，也可以根据教学目标，灵活地调整章节的顺序，增减各章的学时数。

在本书成书的过程中，得到了万昭祎、李旭、苏俊洋以及阿里巴巴的李妹芳等人的大力支持，在此表示衷心的感谢。

由于大数据是一门新兴学科，大数据的教学方法本身还在探索之中，加之我们的水平和能力有限，本书难免有疏漏之处。恳请各位同仁和广大读者给予批评指正，也希望各位能将实践过程中的经验和心得与我们交流(yunxianglu@hotmail.com)。

作　者

2017 年 1 月

目录

CONTENTS

第一部分 大数据概述及基础

第三部分 大数据分析与应用

第一部分

大数据概述及基础

第1章 大数据概念和发展背景

1.1 什么是大数据

大数据是一个不断发展的概念，可以指任何体量或复杂性超出常规数据处理方法的处理能力的数据。数据本身可以是结构化、半结构化甚至是非结构化的，随着物联网技术与可穿戴设备的飞速发展，数据规模变得越来越大，内容越来越复杂，更新速度越来越快，大数据研究和应用已成为产业升级与新产业崛起的重要推动力量。

从狭义上讲，大数据主要是指处理海量数据的关键技术及其在各个领域中的应用，是指从各种组织形式和类型的数据中发掘有价值的信息的能力。一方面，狭义的大数据反映的是数据规模之大，以至于无法在一定时间内用常规数据处理软件和方法对其内容进行有效的抓取、管理和处理；另一方面，狭义的大数据主要是指海量数据的获取、存储、管理、计算分析、挖掘与应用的全新技术体系。

从广义上讲，大数据包括大数据技术、大数据工程、大数据科学和大数据应用等与大数据相关的领域。大数据工程是指大数据的规划、建设、运营、管理的系统工程；大数据科学主要关注大数据网络发展和运营过程中发现和验证大数据的规律及其与自然和社会活动之间的关系。

1.2 大数据的特点

学术界已经总结了大数据的许多特点，包括体量巨大、速度极快、模态多样、潜在价值大等。

IBM 公司使用 3V 来描述大数据的特点。

(1) Volume(体量)。通过各种设备产生的海量数据体量巨大,远大于目前互联网上的信息流量。

(2) Variety(多样)。大数据类型繁多,在编码方式、数据格式、应用特征等多个方面存在差异,既包含传统的结构化数据,也包含类似于 XML、JSON 等半结构化形式和更多的非结构化数据;既包含传统的文本数据,也包含更多的图片、音频和视频数据。

(3) Velocity(速率)。数据以非常高的速率到达系统内部,这就要求处理数据段的速度必须非常快。

后来,IBM 公司又在 3V 的基础上增加了 Value(价值)维度来表述大数据的特点,即大数据的数据价值密度低,因此需要从海量原始数据中进行分析和挖掘,从形式各异的数据源中抽取富有价值的信息。

IDC 公司则更侧重于从技术角度的考量:大数据处理技术代表了新一代的技术架构,这种架构能够高速获取和处理数据,并对其进行分析和深度挖掘,总结出具有高价值的数据。

大数据的“大”不仅仅是指数据量的大小,也包含大数据源的其他特征,如不断增加的速度和多样性。这意味着大数据正以更加复杂的格式从不同的数据源高速向我们涌来。

大数据有一些区别于传统数据源的重要特征,不是所有的大数据源都具备这些特征,但是大多数大数据源都会具备其中的一些特征。

大数据通常是由机器自动生成的,并不涉及人工参与,如引擎中的传感器会自动生成关于周围环境的数据。

大数据源通常设计得并不友好,甚至根本没有被设计过,如社交网站上的文本信息流,我们不可能要求用户使用标准的语法、语序等。

因此大数据很难从直观上看到蕴藏的价值大小,所以创新的分析方法对于挖掘大数据中的价值尤为重要,更是迫在眉睫。

1.3 大数据的发展

大数据技术是一种新一代技术和构架,它成本较低,以快速的采集、处理和分析技术从各种超大规模的数据中提取价值。大数据技术不断涌现和发展,让我们处理海量数据更加容易、便宜和迅速,成为利用数据的好助手,甚至可以改变许多行业的商业模式。大数据技术的发展可以分为六大方向。

(1) 大数据采集与预处理方向。这个方向最常见的问题是数据的多源和多样性，导致数据的质量存在差异，严重影响到数据的可用性。针对这些问题，目前很多公司已经推出了多种数据清洗和质量控制工具(如 IBM 公司的 Data Stage)。

(2) 大数据存储与管理方向。这个方向最常见的挑战是存储规模大，存储管理复杂，需要兼顾结构化、非结构化和半结构化的数据。分布式文件系统和分布式数据库相关技术的发展正在有效地解决这些方面的问题。在大数据存储和管理方向，尤其值得我们关注的是大数据索引和查询技术、实时及流式大数据存储与处理的发展。

(3) 大数据计算模式方向。由于大数据处理多样性的需求，目前出现了多种典型的计算模式，包括大数据查询分析计算(如 Hive)、批处理计算(如 Hadoop MapReduce)、流式计算(如 Storm)、迭代计算(如 HaLoop)、图计算(如 Pregel)和内存计算(如 HANA)，这些计算模式的混合计算方法将成为满足多样性大数据处理和应用需求的有效手段。

(4) 大数据分析与挖掘方向。在数据量迅速增加的同时，还要进行深度的数据分析和挖掘，并且对自动化分析要求越来越高。越来越多的大数据分析工具和产品应运而生，如用于大数据挖掘的 RHadoop 版、基于 MapReduce 开发的数据挖掘算法等。

(5) 大数据可视化分析方向。通过可视化方式来帮助人们探索和解释复杂的数据，有利于决策者挖掘数据的商业价值，进而有助于大数据的发展。很多公司也在开展相应的研究，试图把可视化引入其不同的数据分析和展示的产品中，各种可能相关的商品将会不断出现。可视化工具 Tableau 的成功上市反映了大数据可视化的需求。

(6) 大数据安全方向。当我们在用大数据分析和数据挖掘获取商业价值的时候，黑客很可能在向我们攻击，收集有用的信息。因此，大数据的安全一直是企业和学术界非常关注的研究方向。文件访问控制权限 ACL、基础设备加密、匿名化保护技术和加密保护等技术正在最大程度地保护数据安全。

1.4　大数据的应用

大数据在各行各业的应用越来越频繁与深入，接下来将以几个具体的例子讲述大数据在行业中的应用。

(1) 梅西百货的实时定价机制。根据需求和库存的情况，该公司基于 SAS 的系统对多达 7300 万种货品进行实时调价。

(2) Tipp24 AG 针对欧洲博彩业构建的下注和预测平台。该公司用 KXEN 软件来分析数十亿计的交易以及客户的特性，然后通过预测模型对特定用户进行动态的营销活动。这项举措减少了 90%的预测模型构建时间。

(3) 沃尔玛的搜索。这家零售业寡头为其网站 Walmart.com 自行设计了最新的搜索引擎 Polaris，利用语义数据进行文本分析、机器学习和同义词挖掘等。根据沃尔玛的说法，语义搜索技术的运用使得在线购物的完成率提升了 10%到 15%。

(4) 快餐业的视频分析。其主要通过视频分析等候队列的长度，然后自动变化电子菜单显示的内容。如果队列较长，则显示可以快速供给的食物；如果队列较短，则显示利润较高但准备时间相对长的食品。

(5) PredPol 和预测犯罪。PredPol 公司通过与洛杉矶和圣克鲁斯的警方以及一群研究人员合作，基于地震预测算法的变体和犯罪数据来预测犯罪发生的几率，可以精确到 500 平方英尺的范围内。在洛杉矶运用该算法的地区，盗窃罪和暴力犯罪分别下降了 33%和 21%。

(6) Tesco PLC(特易购)和运营效率。这家连锁超市在其数据仓库中收集了 700 万部冰箱的数据。通过对这些数据的分析进行更全面的监控，并进行主动的维修以降低整体能耗。

(7) American Express(美国运通，AmEx)和商业智能。以往，AmEx 只能实现事后诸葛式的报告和滞后的预测。专家 Laney 认为，“传统的 BI 已经无法满足业务发展的需要。”于是，AmEx 开始构建真正能够预测忠诚度的模型，基于历史交易数据，用 115 个变量进行分析预测。该公司表示，通过预测，对于澳大利亚将于此后的 4 个月中流失的客户已经能够识别出 24%。

1.5 习题

1. 从广义与狭义两方面描述你理解的大数据。
2. 简要陈述大数据的 4V 特点。
3. 简要描述大数据发展的六大方向。
4. 结合现实陈述两个大数据的应用场景。

第 2 章

大数据系统架构概述

这里讲的系统架构设计指的是企业大数据系统设计。深处时代变革中的企业又一次面临大数据这一信息技术革命带来的冲击,企业要么积极拥抱变化,提前做出变革;要么静观其变,择机而动。不管选择哪种方式,都是继互联网之后对企业的又一次智慧的考验。

企业信息化涉及企业的各个方面,是一项复杂的系统工程,一般要经历从初始到不断成熟的过程。对于数据管理阶段和成熟阶段,美国管理信息系统专家诺兰发表了著名的企业信息系统进化的阶段模型,即诺兰模型。诺兰认为,在数据管理阶段,企业高层已经意识到了企业信息战略的重要性,并开始着手企业信息资源的统一规划;在数据成熟阶段,企业和数据是同步发展的,数据是企业面貌的镜像,企业可以依据数据做出发展决策。

2.1 总体架构概述

2.1.1 总体架构设计原则

企业级大数据应用框架需要满足业务的需求:一是要求能够满足基于数据容量大、数据类型多、数据流通快的大数据基本处理需求,能够支持大数据的采集、存储、处理和分析;二是要能够满足企业级应用在可用性、可靠性、可扩展性、容错性、安全性和保护隐私等方面的基本准则;三是要能够满足用原始技术和格式来实现的数据分析的基本

要求。

1. 满足大数据的3V要求

1）大数据容量的加载、处理和分析

要求大数据应用平台经过扩展可以支持GB、TB、PB、EB甚至ZB规模的数据集。

2）各种类型数据的加载、处理和分析

支持各种各样的数据类型，支持处理交易数据、各种非结构化数据、机器数据以及其他新数据结构；支持极端的混合工作负载，包括数以千计的地理上分布的在线用户和程序，这些用户和程序执行各种各样的请求，范围从临时性的请求到战略分析的请求，同时以批量或流的方式加载数据。

3）大数据的处理速度

在很高速度（GB/s）的加载过程中集成来自多个来源的数据；以至少每秒千兆字节的速度高速加载数据，随时进行分析；以满负荷速度就地更新数据；不需要预先将维表与事实表群集即可将十亿行的维表加入万亿行的事实表；在传入的加载数据上实时执行某些“流”分析查询。

2. 满足企业级应用的要求

1）高可扩展性

要求平台符合企业未来业务发展要求以及对新业务的响应，能够支持大规模数据计算的节点可扩展，能适应将来数据结构的变化、数据容量增长、用户的增加、查询要求和服务内容的变化，要求大数据架构具备支持调度和执行数百上千节点的负载工作流。

2）高可用性

要求平台能够具备实时计算环境所具备的高可用性，在单点故障的情况下能够保证应用的可用性，具备处理节点故障时的故障转义和流程继续的能力。

3）安全性和保护隐私

系统在数据采集、存储、分析架构上保证数据、网络、存储和计算的安全性，具备保护个人和企业隐私的措施。

4）开放性

要求平台能够支持计算和存储数以千计的、地理位置可能不同的、可能异构的计算节点，能够识别和整合不同技术和不同厂商开发的工具和应用，能够支持移动应用、互联网应用、社交网络、云计算、物联网、虚拟化、网络、存储等多种计算机设备、计算协议和计算架构。

5）易用性

系统功能操作是否易用，能否满足大多数企业业务、管理和技术人员的操作习惯；平台具有可编程性，能够支持不同编程工具和语言的集成，具备集成编译环境；能否在处理请求内嵌入任意复杂的用户定义函数（UDF），以各种行业标准过程语言执行 UDF，组合大部分或全部使用案例的大量可复用 UDF 库，在几分钟内对 PB 级别大小的数据集执行 UDF“关系扫描”。

3. 满足对原始格式数据进行分析的要求

系统具备对复杂的原始格式数据进行整合分析的能力，如对文本数据、数学数据、统计数据、金融数据、图像数据、声音数据、地理空间数据、时序数据、机器数据等进行分析的能力。

2.1.2　总体架构参考模型

基于 Apache 开源技术的大数据平台总体架构参考模型如图 2-1 所示，大数据的产生、组织和处理主要是通过分布式分拣处理系统来实现的，主流的技术是 Hadoop＋MapReduce，其中 Hadoop 的分布式文件处理系统（HDFS）作为大数据存储的框架，分布式计算框架 MapReduce 作为大数据处理的框架。

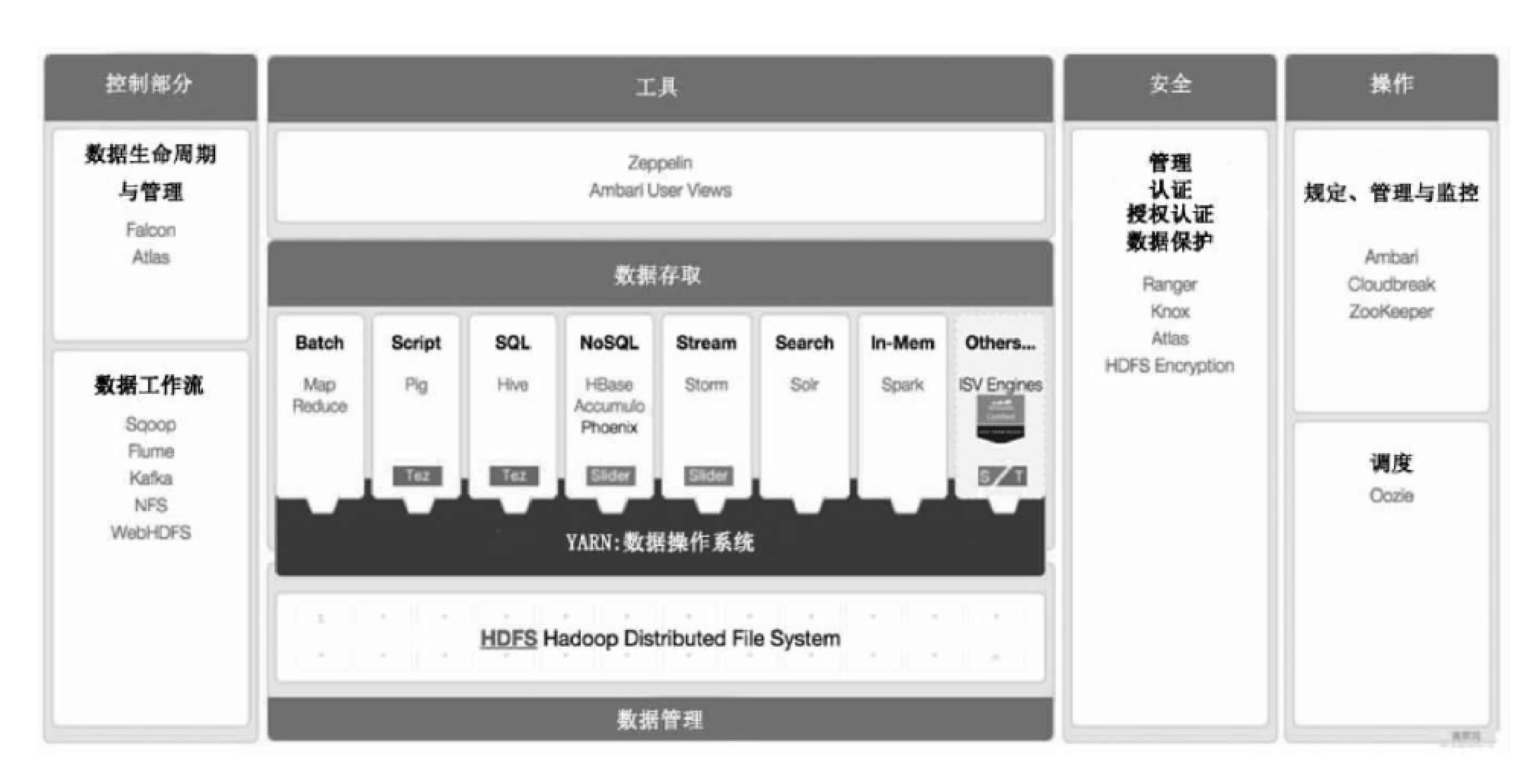


图 2-1　大数据应用平台的总体架构参考模型

1. 大数据基础

这一部分提供了大数据框架的基础，包括序列化、分布式协同等基础服务，构成了上

层应用的基础。

（1）Avro。新的数据序列化与传输工具，将逐步取代 Hadoop 原有的 IPC 机制。

（2）ZooKeeper。分布式锁设施，它是一个分布式应用程序的集中配置管理器，用户分布式应用的高性能协同服务，由 Facebook 贡献，也可以独立于 Hadoop 使用。

2. 大数据存储

HDFS 是 Hadoop 分布式文件系统，前面已经介绍过，HDFS 运行于大规模集群之上，集群使用廉价的普通机器构建，整个文件系统采用的是元数据集中管理与数据块分散存储相结合的模式，并通过数据的冗余复制来实现高度容错。分布式文件处理系统架构在通用的服务器、操作系统或虚拟机上。

3. 大数据处理

MapReduce 是分布式并行计算框架，是基于 Map（可理解为“任务分解”）和 Reduce（可理解为“结果综合”）的函数。基于 MapReduce 写出的应用程序能够运行在由上千个普通机器组成的大型集群上，并以一种可靠容错的方式并行处理 TB 级别以上的数据集。Mapper 和 Reducer 的主代码可以用很多语言编写，Hadoop 的原生语言是 Java，但是 Hadoop 公开 API 用于以 Ruby 和 Python 等其他语言编写代码，还提供了 C++接口。在最底层进行 MapReduce 编程显然提供了最大的潜力，但这种编程层次非常像汇编语言的编程。

4. 大数据访问和分析

在 Hadoop＋MapReduce 之上架构的是基础平台服务，在基础平台之上是大数据访问和分析的应用服务。大数据访问和分析的框架实现对传统关系型数据库和 Hadoop 的访问，主流技术包括 Pig、Hive、Sqoop、Mahout 等。

（1）Pig。Pig 是基于 Hadoop 的并行计算高级编程语言，它提供一种类似于 SQL 的数据分析高级文本语言，称为 Pig Latin，该语言的编译器会把类 SQL 的数据分析请求转换为一系列经过优化处理的 MapReduce 运算。Pig 支持的常用数据分析主要有分组、过滤、合并等，Pig 为创建 Apache MapReduce 应用程序提供了一款相对简单的工具，它有效简化了编写、理解和维护程序的工作，还优化了任务自动执行的功能，并支持使用自定义功能进行接口扩展。

（2）Hive。Hive 是由 Facebook 贡献的数据仓库工具，是 MapReduce 实现的用来查询分析结构化数据的中间件。Hive 的类 SQL 查询语言——Hive SQL 可以查询和分析储存在 Hadoop 中的大规模数据。

(3) Sqoop。Sqoop 由 Cloudera 开发，是一种用于在 Hadoop 与传统数据库间进行数据传递的开源工具，允许将数据从关系源导入 HDFS 以及从 HDFS 导出到关系型数据库。MapReduce 等函数都可以使用由 Sqoop 导入 HDFS 中的数据。

(4) Mahout。Apache Mahout 项目提供分布式机器学习和数据挖掘库。

(5) Hama。基于 BSP 的超大规模科学计算框架。

2.2 运行架构概述

运行架构设计着重考虑的是企业大数据系统运行期的质量属性，比如性能、可伸缩性和持续可用性。大规模用户并发和海量数据处理是企业大数据系统在运行架构设计时重点要解决的问题。

2.2.1 物理架构

企业大数据系统的各层次系统最终要部署到主机节点中，这些节点通过网络连接成为一个整体，为企业的大数据应用提供物理支撑。如前文所述，企业大数据系统由多个逻辑层组成，多个逻辑层可以映射到一个物理节点上，也可以映射到多个物理节点上。

在映射时需要考虑 3 个方面的问题：一是是否容易识别，即通过物理节点的 IP 地址就能知道这个节点的作用域，通过多个物理节点的 IP 地址就能知道这些节点是否为一个集群的；二是是否足够集约，对于负载轻的系统，如果每一个软件系统单独部署在一个物理节点，会造成物理节点的浪费；三是是否能够同构，对于物理节点最好能够统一配置，不仅便于统一管理，而且还可以实现重用，只需一次配置，多个物理节点同构复制，就可以实现动态扩展。

Google 和 Facebook 公司都采用大量的廉价商用硬件来搭建自己的分布式系统，基于廉价商用硬件搭建的分布式系统在运行效率、可靠性、可扩展性方面都被证明能够经得起大规模、高并发、海量数据的检验。

2.2.2 集成架构

企业大数据系统由多个系统集成而成，每个系统都提供了多种协议和接口，以便企业大数据系统的内部系统间集成和外部系统与大数据系统的集成。

企业大数据系统的集成可以分为总体集成和专项集成，总体集成是指各组成系统间

的集成，通过总体集成可以构成高效、可靠、安全运行的企业大数据系统。若企业大数据系统之外的某个应用系统或大数据系统之内的某个应用系统只想与存储系统、调度系统等进行集成，那么可通过调用这些系统开放的接口来实现，这种集成方式就是专项集成。

在实现总体集成时，应用功能集成的方法是同意以代理系统为核心，各应用系统的功能以 WebService 方式注册在统一代理系统中。统一代理系统既可以作为外部系统与应用系统的中介，为外部系统提供功能服务，同时也可以为内部系统间功能的相互调用提供服务。

应用系统将 WebService 的服务注册到统一应用代理服务器，由统一代理应用系统将其转化成统一的对外 WebServer。应用系统门户等内外部系统通过调用统一的对外 WebService 来向统一代理系统发出服务请求。

2.2.3 安全架构

由于企业大数据系统的数据资源和计算资源广泛地分布在多个节点上，所以用户的身份、权限等安全，数据资源的存储、传输、访问等安全，以及计算资源的访问、监控、调整、恢复等安全，都是企业大数据系统在进行安全架构设计时需要考虑的问题。

一般来讲，企业大数据的安全架构由针对 3 层的安全设计构成，这 3 层分别是用户层、应用层和数据层。针对每一层的关键行为加入安全因素的设计，以确保系统的整体安全。

用户层的安全主要是指用户身份安全和用户权限安全，主要由统一代理系统来负责。当用户在登录时和登录后访问应用资源、数据资源时，统一代理系统将对用户身份进行认证，对用户权限进行检查。

用户权限也可以直接将原有的用户权限系统集成到大数据系统中，实现对用户权限的管理，但需要对资源目录进行改造。分布式文件的权限管理粒度到文件级，所以在资源目录中对用户的文件授权也只能到文件级。分布式数据的权限管理粒度只能到行级和列级，而不像传统数据库可以到字段表，所以在资源目录中对用户的数据授权也要做出相应的改变。

应用层安全主要在于能否保证应用安全、可靠地运行。应用层安全关注的行为包括分布式任务提交、进度和状态监管、运行任务的调整、任务的恢复运行、日志记录和资源权限检查。

Hadoop 和 HBase 都提供了相应的机制，以确保应用任务的安全运行。Hadoop 系统通过 JobTracker 来进行 MapReduce 任务的分配、调度和调整，HBase 系统的 HMaster 主节点和 HRegionServer 为了解决数据库中“脏读”和“脏写”的问题会采用 ZooKeeper

的锁服务。

数据层安全重点放在数据是否会丢失、传送过程是否安全、敏感数据是否有加密、数据的完整性是否被破坏 4 个方面。

对于 HDFS 而言，每一个文件的数据库都采用了多副本机制，并将这些副本都保存在不同的节点上。当某个节点的副本失效时，HDFS 还会在一个新的节点上复制一个副本，以确保副本数量与设定要求使用一致。在文件的完整性上，HDFS 对每一个块都采用 CRC32 的校验方式来确保数据的完整性。同样，对于分布式数据库 HBase，它提供了类似的分布式数据库安全机制来确保数据不丢失。

为了保障在网络上的传输安全，利用数据加密技术可在一定程度上确保数据在网络传输过程中不会被截取或窃听。SSL（Secure Socket Layer）是为网络通信提供安全及数据完整性保障的一种安全协议，它已被广泛地应用于 Web 浏览器与服务器之间的身份认证和加密传输方面。HDFS 提供有相应的 HTTPS 方式的文件读/写接口，确保数据传输过程的安全。

2.3　阿里云飞天系统体系架构

飞天（Apsara）是由阿里云自主研发、服务全球的超大规模通用计算操作系统。它可以将遍布全球的百万级服务器连成一台超级计算机，以在线公共服务的方式为社会提供计算能力。

从 PC 互联网到移动互联网再到万物互联网，互联网成为世界新的基础设施。飞天希望解决人类计算的规模、效率和安全问题。飞天的革命性在于将云计算的 3 个方向整合起来：提供足够强大的计算能力，提供通用的计算能力，提供普惠的计算能力。

2009 年，阿里云把自己开发的通用计算操作系统命名为“飞天”，是希望通过计算让人类的想象力与创造力得到最大的释放。

7 年过去，飞天已经为全球 200 多个国家和地区的创新创业企业、政府机构等提供服务。

2.3.1　阿里云飞天整体架构

飞天平台的体系架构如图 2-2 所示，整个飞天平台包括飞天内核和飞天开发服务两大部分。

飞天平台内核为上层飞天开发服务提供存储、计算和调度等方面的底层系统支持，

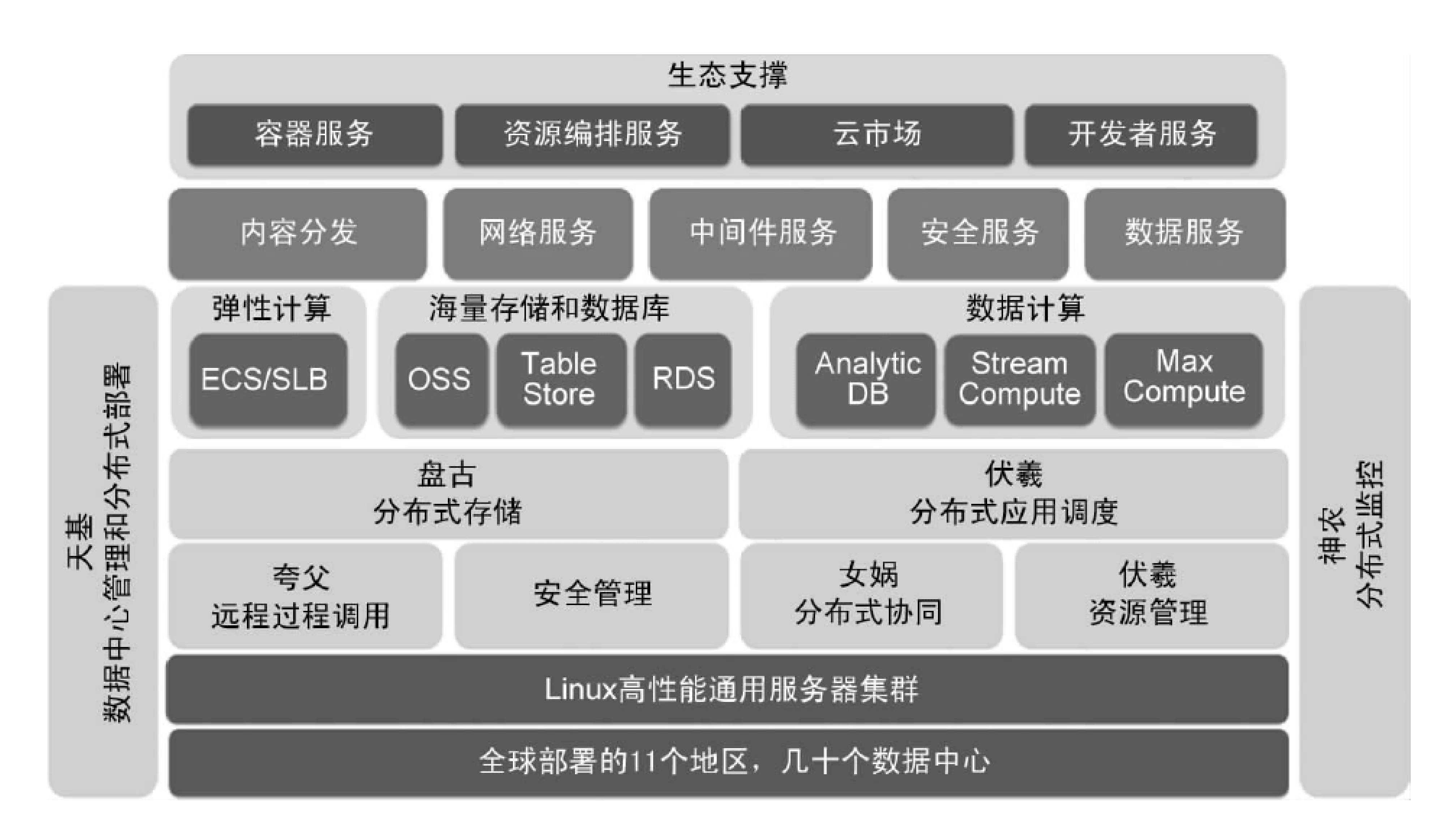


图 2-2　飞天开放平台架构

对应图 2-2 中的盘古分布式存储系统、伏羲分布式应用调度系统、夸父远程过程调用系统、安全管理系统、女娲分布式协同系统、伏羲资源管理系统等模块。飞天开发服务为用户应用程序提供了存储和计算两方面的接口和服务，包括弹性计算 ECS、海量存储和数据库系统（OSS、Table Store、RDS）、数据计算系统（Analytic DB、StreamCompute、MaxCompute）等。

（1）飞天管理着互联网规模的基础设施。其最底层是遍布全球的几十个数据中心和数百个 PoP 节点。飞天所管理的这些物理基础设施还在不断扩张。

（2）飞天内核跑在每个数据中心里面，它负责统一管理数据中心内的通用服务器集群，调度集群的计算、存储资源，支撑分布式应用的部署和执行，并自动进行故障恢复和数据冗余。

（3）安全管理根植在飞天内核最底层。飞天内核提供的授权机制能够有效实现“最小权限原则（principle of least privilege）”，同时还建立了自主可控的全栈安全体系。

（4）监控报警诊断是飞天内核最基本的能力之一。飞天内核对上层应用提供了非常详细的、无间断的监控数据和系统事件采集，能够回溯到发生问题那一刻的现场，帮助工程师找到问题的根源。

（5）在基础公共模块之上有两个最核心的服务，一个叫盘古，一个叫伏羲。盘古是存储管理服务，伏羲是资源调度服务，飞天内核之上应用的存储和资源的分配都是由盘古和伏羲管理。

（6）在基础公共模块旁边还有一个服务，叫天基，意思是“飞天的基础”。天基是飞天的自动化运维服务，负责飞天各个子系统的部署、升级、扩容以及故障迁移。

2.3.2　阿里云飞天平台内核

阿里云飞天平台内核可以分成以下几个部分。

(1) 分布式系统底层服务。其提供分布式环境下所需要的分布式协同服务、远程过程调用服务、安全管理、分布式资源调度等功能,这些服务为上层的分布式文件系统和分布式应用调度提供功能支持。

(2) 盘古分布式文件系统。盘古(Pangu)是一个分布式文件系统,盘古系统的设计目标是将大量通用机器的存储资源聚合在一起,为用户提供大规模、高可靠、高可用、高吞吐量和可扩展的存储服务,将集群中的各个节点的存储能力聚集起来,并能够自动屏蔽硬件故障,为用户提供不间断的数据存取服务;支持增量扩容和数据的自动负载均衡,提供类似于POSIX的用户空间文件访问API,支持随机读/写和追加写的操作,是飞天平台内核中的一个重要组成部分。

(3) 伏羲任务调度系统。该系统为集群中的任务提供调度服务,同时支持强调响应速度的在线服务(Online Service)和强调处理数据吞吐量的离线任务(Batch Processing Job);自动检测系统中的故障和热点,通过错误重试、针对长尾作业并发备份作业等方式保证任务作业可靠、快速地完成。伏羲(Fuxi)是飞天平台内核中负责资源管理和任务调度的模块,同时也为应用开发提供了一套编程基础框架。伏羲同时支持强调响应速度的在线服务和强调处理数据吞吐量的离线任务。

(4) 集群监控和部署。神农(Shennong)是飞天平台内核中负责信息收集、监控和诊断的模块,通过在每台物理机器上部署轻量级的信息采集模块获取各个机器的操作系统与应用软件的运行状态,监控集群中的故障,并通过分析引擎对整个飞天的运行状态进行评估;大禹(Dayu)是飞天内核中负责提供配置管理和部署的模块,它包括一套为集群的运维人员提供的完整工具集,功能涵盖了集群配置信息的集中管理、集群的自动化部署、集群的在线升级、集群扩容、集群缩容,以及为其他模块提供集群基本信息等。

2.3.3　阿里云飞天开放服务

这里简要介绍飞天开放服务,包括弹性计算(ECS)、阿里云对象存储(OSS)、表格存储服务(Table Store)、关系型数据库服务(RDS)、流式计算服务(StreamCompute)和大数据计算服务(MaxCompute)等,这些服务运行在飞天平台内核之上。

(1) 弹性计算(ECS)。云服务器ECS(Elastic Compute Service)是一种云计算服务,它的管理方式比物理服务器更加简单、高效。根据业务的需要,随时创建实例、扩容磁盘

或释放任意多台云服务器实例。

(2) 阿里云对象存储(OSS)。阿里云对象存储(Object Storage Service,OSS)是阿里云对外提供的海量、安全、低成本、高可靠的云存储服务。用户可以通过调用 API 在任何应用、任何时间、任何地点上传和下载数据,也可以通过用户 Web 控制台对数据进行简单的管理。OSS 适合存放任意文件类型,适合各种网站、开发企业及开发者使用,具备高可靠性、安全性、成本低和数据处理能力强等特点。

(3) 表格存储(Table Store)。它是构建在阿里云飞天分布式系统之上的 NoSQL 数据存储服务,提供海量结构化数据的存储和实时访问。表格存储以实例和表的形式组织数据,通过数据分片和负载均衡技术达到规模的无缝扩展;向应用程序屏蔽底层硬件平台的故障和错误,能自动从各类错误中快速恢复,提供非常高的服务可用性;数据全部存储在 SSD 中并具有多个备份,提供了快速的访问性能和极高的数据可靠性。用户在使用表格存储服务时只需要按照预留和使用的资源进行付费,无须关心数据库的软/硬件升级维护、集群缩容/扩容等复杂问题。

(4) 关系型数据库服务(RDS)。阿里云关系型数据库(Relational Database Service,RDS)是一种稳定可靠、可弹性伸缩的在线数据库服务,基于飞天分布式系统和高性能存储,RDS 支持 MySQL、SQL Server、PostgreSQL 和 PPAS(Postgre Plus Advanced Server,一种高度兼容 Oracle 的数据库)引擎,并且提供了容灾、备份、恢复、监控、迁移等方面的全套解决方案,彻底解决数据库运维的烦恼。

(5) 流式计算服务(StreamCompute)。Stream SQL 是 MaxCompute 提供的一种完全托管的分布式数据流式处理服务。该功能底层采用先进的分布式增量计算框架,可以实现低延迟响应,以 SQL 的形式提供流式计算服务,并且完全屏蔽了流式计算中复杂的故障恢复等技术细节,极大地提高了开发效率。

(6) 大数据计算服务(MaxCompute)。大数据计算服务(MaxCompute,原名 ODPS)是一种快速、完全托管的 TB/PB 级数据仓库解决方案。MaxCompute 向用户提供了完善的数据导入方案以及多种经典的分布式计算模型,能够更快速地解决用户的海量数据计算问题,有效降低企业成本,并保障数据安全。

总体来说,飞天核心服务分为计算、存储、数据库和网络。

(1) 为了帮助开发者便捷地构建云上应用,飞天提供了丰富的连接、编排服务,将这些核心服务方便地连接和组织起来,包括通知、队列、资源编排、分布式事务管理等。

(2) 飞天接入层包括数据传输服务、数据库同步服务、CDN 内容分发以及混合云高速通道等服务。

(3) 飞天最顶层是阿里云打造的软件交易与交付第一平台——云市场。它如同云计算的"App Store",用户可在阿里云官网一键开通"软件+云计算资源"。云市场上架

在售商品几千个，支持镜像、容器、编排、API、SaaS、服务、下载等类型的软件与服务接入。

(4) 飞天有一个全球统一的账号体系。灵活的认证授权机制让云上资源可以安全、灵活地在租户内或租户间共享。

通过7年实践，飞天已经建立了一个完善的云产品体系，同时还能提供互联网级别的租户管理和业务支撑服务，并拥有以下核心竞争力和核心能力：

- 自主可控。对云计算底层技术体系的把控力，自主研发，自己解决核心问题。
- 调度能力。10K(单集群一万台服务器)的任务分布式部署和监控。
- 数据能力。EB(10亿GB)级的大数据存储和分析能力。
- 安全能力。为中国35%的网站提供防御。
- 大规模实践。经受双11、12306春运购票等极限并发场景挑战。
- 开放的生态。兼容大多数生态软件和硬件，比如CLoudfudry、Docker、Hadoop。

2.3.4 阿里云飞天的特色

1. 阿里云飞天OpenStack和Hadoop的不同

飞天是一个操作系统，部署在互联网规模的基础设施之上，它以公共服务的方式对外提供服务，可以让用户直接联网使用，并且提供了丰富的云服务、数据服务以及完善的安全体系和云市场与生态支撑。通过软/硬件一体化，飞天可以实现更优的性能。客户享受的是云上便捷的服务，可以将人力、物力和时间等资源使用在更擅长的领域，从而快速实现商业价值。

OpenStack和Hadoop是软件，它们并没有解决客户的CAPEX投入问题、运维人员投入问题，需要部署到自有的硬件上，一般只用于单个企业的内部环境。从软件变成服务有很多事情需要做，比如说需要把这个软件下载下来，要知道自己的硬件配置是什么，需要部署上去，还有监控，Hadoop生态圈里面虽然有这样那样的解决方案，但是并没有很好地集成起来，这些选择还是需要自己来做。

从软件工程来说，飞天是保持版本迭代稳定性和一致性的一个完整架构，而开源是一个树状分支发展体系，无法保障整体诉求的统一性。

飞天上面提供了基于Hadoop、EMR、Mongo等开源软件的托管服务，这是飞天开放能力的体现。

2. 阿里云飞天与VMware、华为FusionSphere的不同

虚拟化不等于云计算，云的实时在线、海量弹性、多租户隔离、专业运维都是传统虚

拟化软件所欠缺的。

飞天是一个操作系统，部署在互联网规模的基础设施之上，它以公共服务的方式对外提供服务，并且提供了丰富的元数据服务、数据服务以及完善的安全体系和云市场及生态支撑。客户享受的是云上便捷的服务，没必要将人力、物力和时间成本浪费在不擅长的领域，从而快速实现商业价值。

而 VMware（ESX、NSX、VSAN）以及华为 FusionSphere 都是软件，并没有解决客户的 CAPEX 投入问题、运维人员投入问题，一般只用于单个企业的内网环境，没有完善的多租户体系和生态体系。

VMware 的三大件主要解决了计算的效率问题，但是没有解决计算的规模问题。

华为的 FusionSphere 其实是基于开源软件进行定制并适配华为硬件的软件系统，飞天内核在规模、性能、稳定性和通用性上都超越了 FusionSphere。

2.4 主流大数据系统厂商

如今，越来越多的企业和大型机构在寻求不断发展的大数据问题时都倾向于使用开源软件基础架构 Hadoop 的服务，因此许多公司推出了各自版本的 Hadoop，也有一些公司围绕 Hadoop 开发产品。本节将以列举主流厂商解决方案的方式呈现不同厂商的处理异同。

2.4.1 阿里云数加平台

数加平台是阿里云为企业大数据的实施提供的一套完整的一站式大数据解决方案，覆盖了企业数据仓库、商业智能、机器学习、数据可视化等领域，助力企业在 DT 时代更敏捷、更智能、更具洞察力。

数加平台由大数据计算服务（MaxCompute）、分析型数据库（AnalyticDB）、流计算（StreamCompute）共同组成了底层强大的计算引擎，速度更快，成本更低。在计算引擎之上，数加平台提供了丰富的云端数据开发套件，包括数据集成、数据开发、调度系统、数据管理、运维视屏、数据质量、任务监控等。

数加平台的整体架构如图 2-3 所示。

数加平台有以下优势。

（1）一站式大数据解决方案。从数据导入、查找、开发、ETL、调度、部署、建模、BI 报表、机器学习到服务开发、发布，以及外部数据交换的完整大数据链路，一站式集成开发环境，降低数据创新与创业成本，如图 2-4 所示。

（2）大数据与云计算的无缝结合。阿里云数加平台构建在阿里云的云计算基础设施

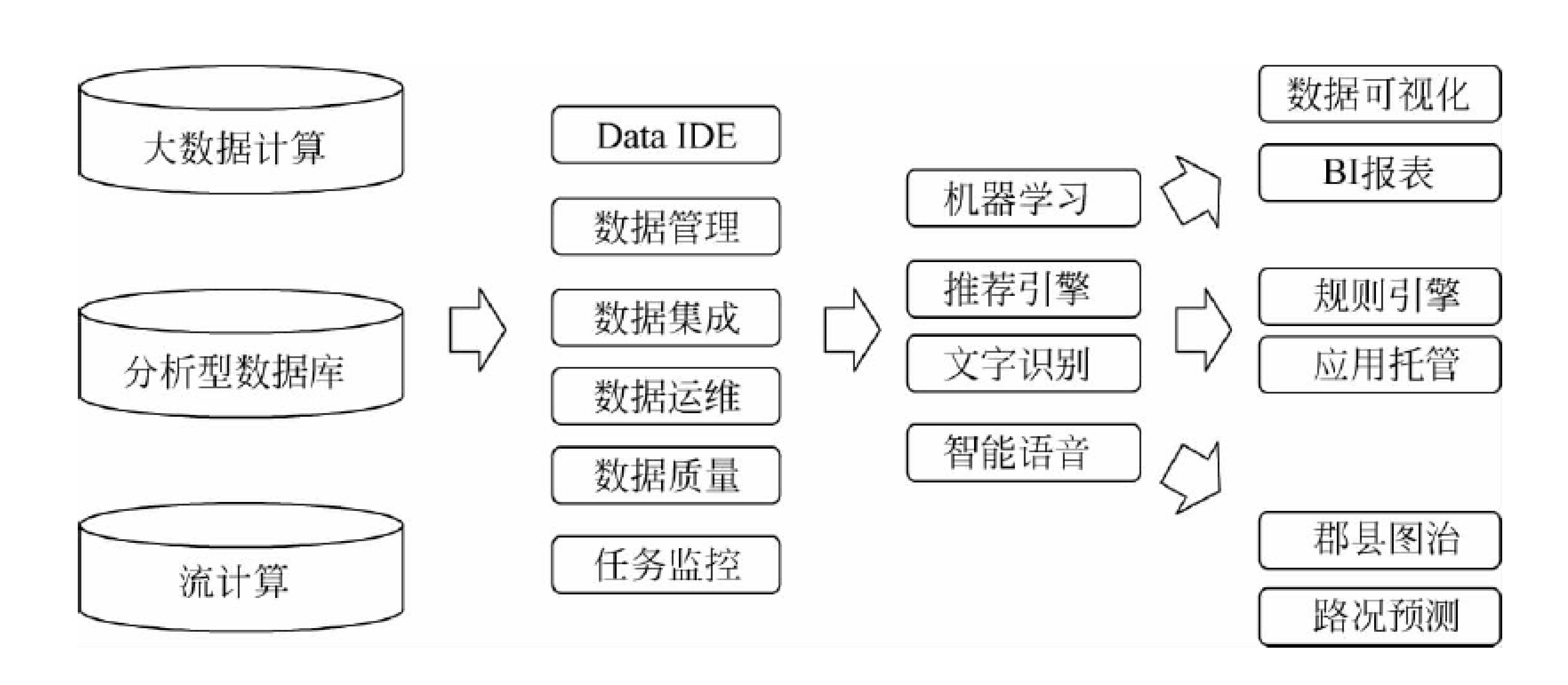


图 2-3　阿里云数加平台架构

之上，简单、快速地接入 MaxCompute 等计算引擎，支持 ECS、RDS、OCS、AnalyticDB 等云设施下的数据同步，为企业获得在大数据时代最重要的竞争力——智能化。

（3）企业级数据安全控制。数加平台建立在安全性业界领先的阿里云上，并集成了最新的阿里云大数据产品，这些大数据产品的性能和安全性在阿里巴巴集团内部已经得到多年的锤炼。数加平台采用了先进的“可用不可见”的数据合作方式，并对数据所有者提供全方位的数据安全服务，数据安全体系包括数据业务安全、数据产品安全、底层数据安全、云平台安全、接入和网络安全、运维管理安全。



图 2-4　一站式解决方案

2.4.2　Cloudera

Cloudera 是一家专业从事基于 Apache Hadoop 的数据管理软件销售和服务的公司，它发布的实时查询开源项目 Impala 比基于 MapReduce 的 Hive SQL 的查询速度提升了 3～90 倍。Impala 是 Google Dremel 的模仿，但在 SQL 功能上更胜一筹，而且使用简单、灵活。

Cloudera Impala 对存储在 Apache Hadoop HDFS、HBase 的数据提供直接查询互动的 SQL，既可以像 Hive 使用相同的统一存储平台，Impala 也使用相同的元数据、SQL 语法（Hive SQL）、ODBC 驱动程序和用户界面（Hue Beeswax）。Impala 还提供了一个面向批量或实时查询的统一平台。

Flume 是 Cloudera 提供的一个高可用性、高可靠性、分布式的海量日志采集、聚合和传输的系统，它支持在日志系统中定制各类数据发送方，用于收集数据；同时，Flume 提供对数据进行简单处理并写到各种数据接受方（可定制）的能力。

Flume 提供了从 console（控制台）、RPC（Thrift-RPC）、text（文件）、tail（UNIX tail）、syslog（syslog 日志系统，支持 TCP 和 UDP 两种模式）、exec（命令执行）等数据源上收集数据的能力。

Flume 采用了多 Master 的方式。为了保证配置数据的一致性，其引入了 ZooKeeper，用于保存配置数据，ZooKeeper 本身可保证配置数据的一致性和高可用。另外，在配置数据发生变化时，ZooKeeper 可以通知 Flume 的 Master 节点。Flume Master 间使用 Gossip 协议对数据进行同步。

2.4.3 Hortonworks

Hortonworks 的开放式互联平台能帮助企业管理所拥有的数据（动态数据以及静态数据），为用户组织启用可操作情报。

HDP（Hortonworks Data Platform）是一款基于 Apache Hadoop 的开源数据平台，提供大数据云存储、大数据处理和分析等服务。该平台专门用来应对多来源和多格式的数据，并使其处理起来能变得更加简单、更有成本效益。

HDP 还提供了一个开放、稳定和高度可扩展的平台，使其更容易地集成 Apache Hadoop 的数据流业务与现有的数据架构。该平台包括各种 Apache Hadoop 项目以及 Hadoop 分布式文件系统（HDFS）、MapReduce、Pig、Hive、HBase、ZooKeeper 和其他各种组件，使 Hadoop 的平台更易于管理，更加具有开放性以及可扩展性。

2.4.4 Amazon

Amazon 公司的 AWS 本身就是最完整的大数据平台，Amazon Web Services 提供了一系列广泛的服务，可以快速、轻松地构建和部署大数据分析应用程序。借助 AWS 可以迅速扩展几乎任何大数据应用程序，其中包括数据仓库、点击流分析、欺诈侦测、推荐引擎、事件驱动 ETL、无服务器计算和物联网处理等应用程序。

Amazon EMR 是一种 Web 服务，旨在实现轻松快速并经济高效地处理大量的数据。

Amazon EMR 提供托管的 Hadoop 框架，可以在多个支持动态扩展的 Amazon EC2 实例之间分发和处理大量数据。这里的 Hadoop 也可以替换为其他常用的分发框架，例如 Spark 或 Presto。同时框架中的文件系统可以使用 AWS 数据服务代替，例如 Amazon S3 和 Amazon DynamoDB，用于进行数据交换。Amazon EMR 能够安全、可靠地处理大数据使用案例，包括日志分析、Web 索引、数据仓库、机器学习、财务分析等，还原生支持了全文索引 ElasticSearch。

2.4.5 Google

Google 公司作为大数据研究的引领者，为大数据的研究和应用提供了大量的论文和实现。其中 Google 文件系统(Google File System，GFS)作为 Google 大数据存储与处理的基石，其开源实现 HDFS 也是 Hadoop 的关键组件。GFS 采用大量的低可靠性 PC 构成集群系统的思想也为后来的大数据系统所继承。GFS 采用"主控"服务器、"Chunk"服务器与客户端的架构，实现了分布式存储对应用开发者的透明化，使其类似于本地的文件系统，这一机制也广泛被各种分布式文件系统所采用。

不止如此，Google 提出的 MapReduce 计算框架在很多大数据领域得到了非常广泛的应用；Google 研发的针对分布式系统协调管理的粗粒度锁服务 Chubby 实现了一个实例对上万台机器的协同管理；Google 针对微服务架构提出的 GRC 远程调用框架实现了分布式系统对不同语言和框架的兼容，让新的编程模型——微服务架构的实际应用成为了现实。

可以说，Google 在大数据领域拥有最多的成熟解决方案，也对大数据技术的发展起到了非常重要的推动作用。

2.4.6 微软

微软公司推出的商业数据分析系统 Microsoft Analytics Platform System 能够通过其扩充的大规模平行处理整合式系统支持混合格式的数据仓库，借此适应数据仓库环境不断发展的需求。它能够运用 Microsoft PolyBase 和从 SQL Server 以来积累的海量数据处理技术，在关系式和非关系式数据库中进行查询。

此外，微软还提供了基于 Hadoop 的分布式解决方案 Microsoft Azure，其中最值得注意的是"云端 Hadoop"——HDInsight，它提供了一系列全面的 Apache 大数据项目的托管服务。Azure HDInsight 使用 Hortonworks Data Platform (HDP)分布式 Hadoop。

HDInsight 在云上部署 Hadoop 集群，并提供管理服务和一个处理、分析以及报告大数据的高稳定性和可用性框架。HDInsight 同时还支持 Apache 的 Storm 平台，以提供即时监控和串流数据分析。

2.5 习题

1. 简要介绍大数据访问框架的主流实现技术。
2. 介绍大数据系统运行框架设计的组成部分。
3. 简述大数据系统物理架构设计中映射过程需要考虑的问题。
4. 简述大数据安全架构设计针对的 3 层内容。

第3章

分布式通信与协同

在大规模分布式系统中，为了高效地处理大量任务以及存储大量数据，通常需要涉及多个处理节点，需要在多个节点之间通信以及协同处理。高效的节点之间的通信以及节点之间的可靠协同技术是保证分布式系统正常运行的关键。

3.1 数据编码传输

3.1.1 数据编码概述

在分布式系统中需要处理大量的网络数据，为了加快网络数据的传输速度，通常需要对传输数据进行编码压缩，当然数据编码压缩传输技术也在其他电子信息领域中大量使用，由于数字化的多媒体信息尤其是数字视频、音频信号的数据量特别庞大，如果不对其进行有效的压缩难以得到实际的应用，因此数据编码压缩技术已成为当今数字通信、广播、存储和多媒体娱乐中的一项关键的共性技术。

数据压缩是以尽可能少的数码来表示信源所发出的信号，减少容纳给定的消息集合或数据采样集合的信号空间。这里讲的信号空间就是被压缩的对象，是指某信号集合所占的时域、空域和频域。信号空间的这几种形式是相互关联的，存储空间的减少意味着信号传输效率的提高，所占用带宽的节省。只要采取某种方法来减少某个信号空间就能够压缩数据。

一般来说，数据压缩主要是通过数据压缩编码来实现的。要想使编码有效，必须

建立相应的系统模型。在给定的模型下通过数据编码来消除冗余，大致有以下 3 种情况。

（1）信源符号之间存在相关性。如果消除了这些相关性，就意味着数据压缩。例如，位图图像像素与像素之间的相关性，动态视频帧与帧之间的相关性。去掉这些相关性通常采用预测编码、变换编码等方法。

（2）信源符号之间存在分布不等概性。根据不同符号出现的不同概率分别进行编码，概率大的符号用较短的码长编码，概率小的符号用较长的码长编码，最终使信源的平均码长达到最短。通常采用统计编码的方法。

（3）利用信息内容本身的特点（如自相似性）。用模型的方法对需传输的信息进行参数估测，充分利用人类的视觉、听觉等特性，同时考虑信息内容的特性，确定并遴选出其中的部分内容（而不是全部内容）进行编码，从而实现数据压缩。通常采用模型基编码的方法。

目前比较认同的、常用的数据压缩的编码方法大致分为两大类。

（1）冗余压缩法或无损压缩法。冗余压缩法或无损压缩法又称为无失真压缩法或熵编码法。这类压缩方法只是去掉数据中的冗余部分，并没有损失熵，而这些冗余数据是可以重新插入到原数据中的。也就是说，去掉冗余不会减少信息量，而且仍可原样恢复数据。因此，这类压缩方法是可逆的。

（2）熵压缩法或有损压缩法。这类压缩法由于压缩了熵，也就损失了信息量，而损失的信息是不能恢复的。因此，在用门限值采样量化时，如果只存储门限内的数据，那么原来超过这个预置门限的数据将丢失。这种压缩方法虽然可压缩大量的信号空间，但那些丢失的实际样值不可能恢复，是不可逆的。也就是说，在用熵压缩法时数据压缩要以一定的信息损失为代价，而数据的恢复只能是近似的，应根据条件和要求在允许的范围内进行压缩。

3.1.2 LZSS 算法

LZSS 算法属于字典算法，是把文本中出现频率较高的字符组合做成一个对应的字典列表，并用特殊代码来表示这个字符。图 3-1 为字典算法原理图示。

LZSS 算法的字典模型使用自适应方式，基本的思路是搜索目前待压缩串是否在以前出现过，如果出现过，则利用前次出现的位置和长度来代替现在的待压缩串，输出该字符串的出现位置及长度；否则，输出新的字符串，从而起到压缩的目的。但是在实际使用过程中，由于被压缩的文件往往较大，一般使用“滑动窗口压缩”方式，也就是说将一个虚拟的、可以跟随压缩进程滑动的窗口作为术语字典。LZSS 算法最大的好处是压缩算法

的细节处理不同，只对压缩率和压缩时间有影响，不会影响到解压程序。LZSS 算法最大的问题是速度，每次都需要向前搜索到原文开头，对于较长的原文需要的时间是不可忍受的，这也是 LZSS 算法较大的一个缺点。

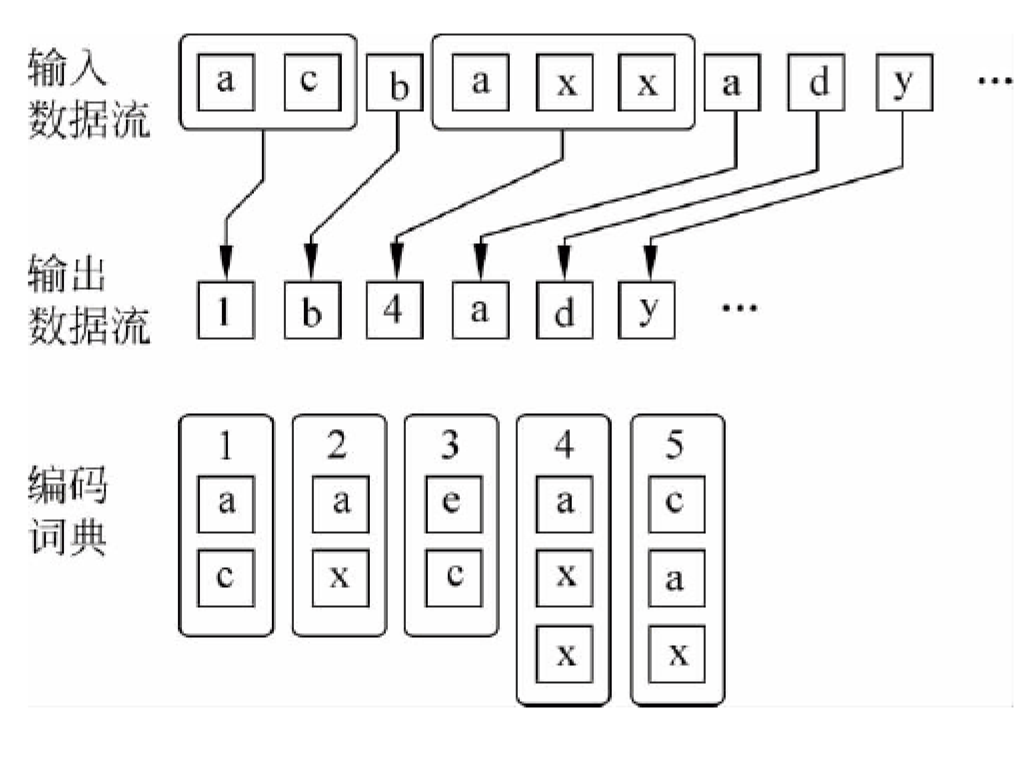


图 3-1 字典算法原理

3.1.3 Snappy 压缩库

Snappy 是在 Google 公司内部生产环境中被许多项目使用的压缩/解压缩的链接库，使用该库的软件包括 BigTable、MapReduce 和 RPC 等，Google 公司于 2011 年开源了该压缩/解压缩库。在 Intel 酷睿 i7 处理器上，在单核 64 位模式下，Snappy 的压缩速度大概可以达到 250MB/s 或者更快，解压缩可以达到大约 500MB/s 甚至更快。如此高的压缩速度是通过降低压缩率来实现的，因此其输出要比其他库大 20%～100%。Snappy 对于纯文本的压缩率为 1.5～1.7，对于 HTML 是 2～4，当然，对于 JPEG、PNG 和其他已经压缩过的数据的压缩率为 1.0。

Snappy 压缩库采用 C++ 实现，同时提供了多种其他语言的接口，包括 C、C#、Go、Haskell 等。Snappy 是面向字节编码的 LZ77 类型压缩器。Snappy 采用的编码单元是字节(byte)，而不是比特(bit)，采用该压缩库压缩后的数据形成一个字节流的格式，格式如下：前面几个字节表示总体为压缩的数据流长度，采用小端方式(little-endian)存储，同时兼顾可变长度编码，每个字节的后面 7 位存储具体的数据，最高位用于表示下一个字节是否为同一个整数；剩下的字节用 4 种元素类型中的一种进行编码，元素类型在元素数据中的第一个字节，该字节的最后 2 位表示类型。

- 00。文本数据，属于未压缩数据，类型字节的高 6 位用于存储每个元素的数据内容长度。当数据内容超过 60 个字节时，采用额外的可变长编码方式存储数据。
- 01。数据长度用 3 位存储，偏移量用 11 位存储。紧接着类型字节后的第一个字节也用于存储偏移量。

- 10。类型字节中剩下的高 6 位用于存储数据长度，在类型字节后的两个字节用于存储数据的偏移量。
- 11。类型字节中剩下的高 6 位用于存储数据长度，数据偏移量存储在类型字节后的 4 个字节，偏移量采用小端方式存储数据。

3.2 分布式通信系统

分布式通信研究分布式系统中不同子系统或进程之间的信息交换机制。我们从各种大数据系统中归纳出 3 种最常见的通信机制：远程过程调用、消息队列和多播通信。其中，远程过程调用的重点是网络中位于不同机器上进程之间的交互；消息队列的重点是子系统之间的消息可靠传递；多播通信是实现信息的高效多播传递。这三者都是黏合子系统的有效工具，同时，它们对于减少大数据系统中构件之间的耦合、增强各自的独立演进有很大的帮助作用。

3.2.1 远程过程调用

远程过程调用(Remote Procedure Call，RPC)是一个计算机通信协议，通过该协议运行于一台计算机上的程序可以调用另一台计算机的子程序，而程序员无须额外地为这个交互编程。

通用的 RPC 框架都支持以下特性：接口描述语言、高性能、数据版本支持以及二进制数据格式。

Thrift 是由 Facebook 公司开发的远程服务调用框架，它采用接口描述语言定义并创建服务，支持可扩展的跨语言服务开发，所包含的代码生成引擎可以在多种语言中，如 C++、Java、Python、PHP、Ruby、Erlang、Perl、Haskell、C#、Cocoa、Smalltalk 等，创建高效的、无缝的服务。其传输数据采用二进制格式，相对于 XML 和 JSON 体积更小，对于高并发、大数据量和多语言的环境更有优势。

Thrift 包含了一个完整的堆栈结构，用于构建客户端和服务器端。服务器包含用于绑定协议和传输层的基础架构，它提供阻塞、非阻塞、单线程和多线程的模式运行在服务器上，可以配合服务器/容器一起运行，可以和现有的服务器/容器无缝结合。

其使用流程大致如下。

首先使用 IDL 定义消息体以及 RPC 函数调用接口。使用 IDL 可以在调用方和被调用方解耦，比如调用方可以使用 C++，被调用方可以使用 Java，这样给整个系统带来了极

大的灵活性。

然后使用工具根据 IDL 定义文件生成指定编程语言的代码。

最后即可在应用程序中连接使用上一步生成的代码。对于 RPC 来说，调用方和被调用方同时引入后即可实现透明的网络访问。

3.2.2　消息队列

消息队列也是设计大规模分布式系统时经常采用的中间件产品。分布式系统构件之间通过传递消息可以解除相互之间的功能耦合，这样减轻了子系统之间的依赖，使得各个子系统或者构建可以独立演进、维护或重用。消息队列是在消息传递过程中保存消息的容器或中间件，其主要目的是提供消息路由并保障消息可靠传递。

下面通过 Linkedin 开源的分布式消息系统 Kafka 介绍消息队列系统的整体设计思路。

Kafka 采用 Pub-Sub 机制，具有极高的消息吞吐量、较强的可扩展性和高可用性，消息传递延迟低，能够对消息队列进行持久化保存，且支持消息传递的“至少送达一次”语义。

一个典型的 Kafka 集群中包含若干 producer、若干 broker、若干 consumer group，以及一个 ZooKeeper 集群。Kafka 通过 ZooKeeper 管理集群配置，选举 leader，以及在 consumer group 发生变化时进行 rebalance。producer 使用 push 模式将消息发布到 broker，consumer 使用 pull 模式从 broker 订阅并消费消息。

作为一个消息系统，Kafka 遵循了传统的方式，选择由 producer 向 broker push 消息并由 consumer 向 broker pull 消息。push 模式很难适应消费速率不同的 consumer，因为消息发送速率是由 broker 决定的。push 模式的目标是尽可能以最快的速度传递消息，但是这样很容易造成 consumer 来不及处理消息，典型的表现就是拒绝服务以及网络阻塞。pull 模式可以根据 consumer 的消费能力以适当的速率消费信息。

3.2.3　应用层多播通信

分布式系统中的一个重要的研究内容是如何将数据通知到网络中的多个接收方，这一般被称为多播通信。与网络协议层的多播通信不同，这里介绍的是应用层多播通信。Gossip 协议就是常见的应用层多播通信协议，与其他多播协议相比，其在信息传递的健壮性和传播效率方面有较好的折中效果，使其在大数据领域中得以广泛使用。

Gossip 协议也被称为“感染协议”(Epidemic Protocol),用来尽快地将本地更新数据通知到网络中的所有其他节点。其具体更新模型又可以分为3种:全通知模型、反熵模型和散步谣言模型。

在全通知模型中,当某个节点有更新消息时立即通知所有其他节点;其他节点在接收到通知后判断接收到的消息是否比本地消息要新,如果是,则更新本地数据,否则,不采取任何行为。反熵模型是最常用的“Gossip 协议”,之所以称之为“反熵”,是因为“熵”是用来衡量系统混乱无序程度的指标,熵越大说明系统越无序。系统中更新的信息经过一定轮数的传播后,集群内的所有节点都会获得全局最新信息,所以系统变得越来越有序,这就是“反熵”的含义。

在反熵模型中,节点 P 随机选择集群中的另一个节点 Q,然后与 Q 交换更新信息;如果 Q 信息有更新,则类似 P 一样传播给任意其他节点(此时 P 也可以再传播给其他节点),这样经过一定轮数的信息交换,更新的信息就会快速传播到整个网络节点。

散步谣言模型与反熵模型相比增加了传播停止判断。即如果节点 P 更新了数据,则随机选择节点 Q 交换信息;如果节点 Q 已经从其他节点处得知了该更新,那么节点 P 降低其主动通知其他节点的概率,直到一定程度后,节点 P 停止通知行为。散布谣言模型能够快速传播变化,但不能保证所有节点都能最终获得更新。

3.2.4 阿里云夸父 RPC 系统

在分布式系统中,不同计算机之间只能通过消息交换的方式进行通信。显式的消息通信必须通过 Socket 接口编程,而 RPC 可以隐藏显式的消息交换,使得程序员可以像调用本地函数一样来调用远程的服务。

夸父(Kuafu)是飞天平台内核中负责网络通信的模块,它提供了一个 RPC 的接口,简化编写基于网络的分布式应用。夸父的设计目标是提供高可用(7×24 小时)、大吞吐量(Gigabyte)、高效率、易用(简明 API、多种协议和编程接口)的 RPC 服务。

RPC 客户端(RPC Client)通过 URI 指定请求需要发送的 RPC 服务端(RPC Server)的地址,目前夸父支持两种协议形式。

- TCP。例如 tcp://fooserver01:9000。
- Nuwa。例如 nuwa://nuwa01/FooServer。

与用流(Stream)传输的 TCP 通信相比,夸父通信是以消息(Message)为单位的,支持多种类型的消息对象,包括标准字符串 std::string 和基于 std::map 实现的若干 string 键-值对。

夸父 RPC 同时支持异步(asynchronous)和同步(synchronous)的远程过程调用形式。

(1) 异步调用。RPC 函数调用时不等接收到结果就会立即返回,用户必须通过显式调用接收函数取得请求结果。

(2) 同步调用。RPC 函数调用时会等待,直到接收到结果才返回。在实现中,同步调用是通过封装异步调用来实现的。

在夸父的实现中,客户端程序通过 UNIX Domain Socket 与本机上的一个夸父代理(Kuafu Proxy)连接,不同计算机之间的夸父代理会建立一个 TCP 连接。这样做的好处是可以更高效地使用网络带宽,系统可以支持上千台计算机之间的互联需求。此外,夸父利用女娲来实现负载均衡;对大块数据的传输作了优化;与 TCP 类似,夸父代理之间还实现了发送端和接收端的流控(Flow Control)机制。

3.2.5 Hadoop IPC 的应用

这里以 Hadoop 中的 RPC 框架 Hadoop IPC 为基础讲述 RPC 框架在大数据系统中的应用。Hadoop 系统包括 Hadoop Common、Hadoop Distributed File System、Hadoop MapReduce 几个重要的组成部分,其中,Hadoop Common 用于提供整个 Hadoop 公共服务,包括 Hadoop IPC。在 Hadoop 系统中,Hadoop IPC 为 HDFS、MapReduce 提供了高效的 RPC 通信机制,在 HDFS 中,DFSClient 模块需要与 NameNode 模块通信、DFSClient 模块需要与 DataNode 模块通信、MapReduce 客户端需要与 JobTracker 通信,Hadoop IPC 为这些模块之间的通信提供了一种便利的方式。

目前实现的 Hadoop IPC 具有采用 TCP 方式连接、支持超时、缓存等特征。Hadoop IPC 采用的是经典的 C/S 结构。

Hadoop IPC 的 Server 端相对比较复杂,包括 Listener、Reader、Handler 和 Responder 等多种类型的线程,Listener 用于侦听来自 IPC Client 端的连接,同时也负责管理与 Client 端之间的连接,包括 Client 端超时需要删除连接;Reader 线程用于读取来自 Client 端的数据,Handler 线程用于处理来自 Client 端的请求,执行具体的操作;Responder 线程用于返回处理结果给 Client 端。一般配置是一个 Listener、多个 Reader、多个 Handler 和一个 Responder。Hadoop IPC 的组成如图 3-2 所示。

执行 HDFS 读文件操作,首先 DFSClient 利用 Hadoop IPC 框架发起一次 RPC 请求给 NameNode,获取 DataBlock 信息。

在执行 HDFS 数据恢复操作的时候,DFSClient 需要执行 recoverBlock RPC 操作,发送该请求到 DataNode 节点上。

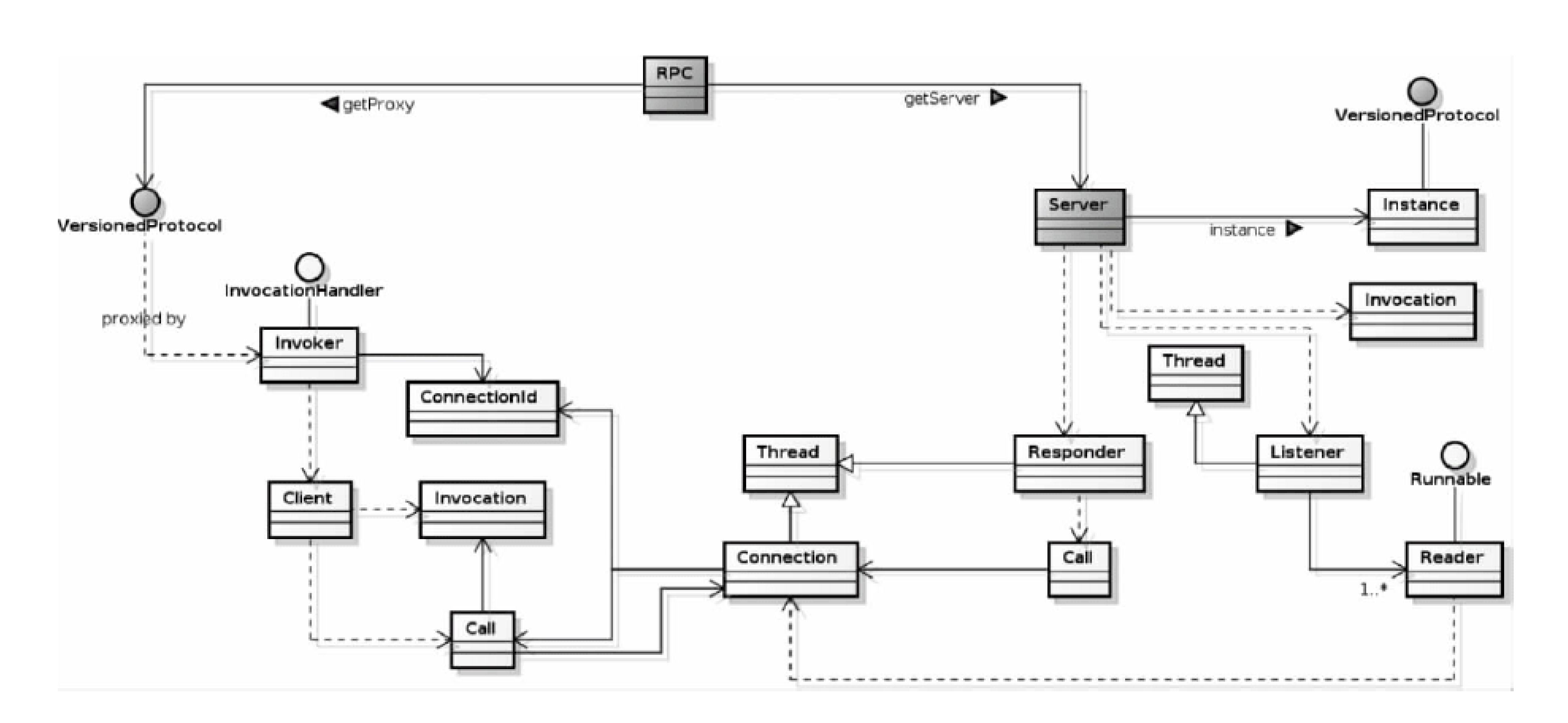


图 3-2 Hadoop IPC 的组成

3.3 分布式协同系统

当前的大规模分布式系统涉及大量的机器,这些机器之间需要进行大量的网络通信以及各个节点之间的消息通信协同。为了减少分布式系统中这些工作的重复开发,解耦出分布式协同系统,有效地提高了分布式计算、分布式存储等系统的开发速度。

3.3.1 Chubby 锁服务

Chubby 是 Google 公司研发的针对分布式系统协调管理的粗粒度服务,一个 Chubby 实例大约可以负责 1 万台 4 核 CPU 机器之间对资源的协同管理。这种服务的主要功能是让众多客户端程序进行相互之间的同步,并对系统环境或资源达成一致的认知。

Chubby 的理论基础是 Paxos(一致性协议),Paxos 是在完全分布式环境下不同客户端能够通过交互通信并投票对于某个决定达成一致的算法。Chubby 以此为基础,但是也进行了改造,Paxos 是完全分布的,没有中心管理节点,需要通过多轮通信和投票来达成最终的一致,所以效率低;Chubby 出于对系统效率的考虑,增加了一些中心管理策略,在达到同一目标的情况下改善了系统效率。

Chubby 的设计目标基于以下几点:高可用性、高可靠性、支持粗粒度的建议性锁服务、支持小规模文件直接存储。这些当然是用高性能与存储能力折中而来的。

图 3-3 是 Google 论文中描述的 Chubby 的整体架构,可以容易地看出 Chubby 共有 5 台服务器,其中一个是主服务器,客户端与服务器之间使用 RPC 交互。那么,其他服务器是干什么的?它们纯粹是作为主服务器不可用后的替代品。而 ZooKeeper 的多余服

务器均是提供就近服务的，也就是服务器会根据地理位置与网络情况来选择对哪些客户端给予服务。

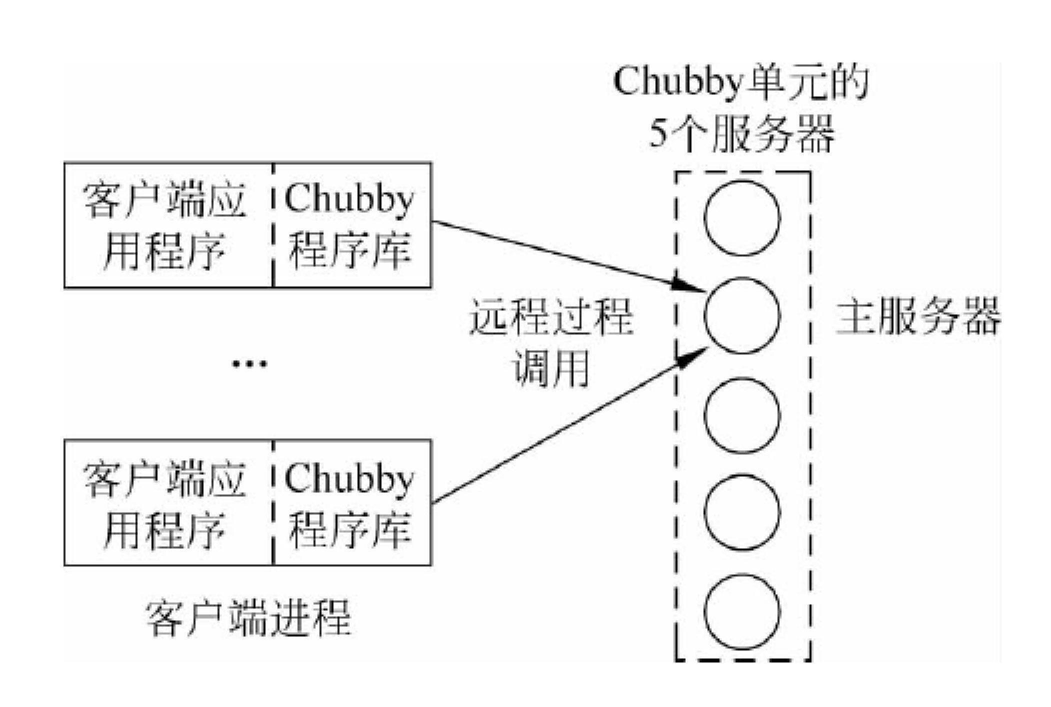


图 3-3　Chubby 整体架构

Chubby 单元中的主服务器由所有服务器选举推出，但是并非从始至终一直都由其担任这一角色，它是有“任期”的，即 Master Lease，一般长达几秒。如果无故障发生，一般系统尽量将“租约”交给原先的主服务器，否则可以通过重新选举得到一个新的全局管理服务器，这样就实现了主服务器的自动切换。

客户端通过嵌入的库程序，利用 RPC 通信和服务器进行交互，对 Chubby 的读/写请求都由主服务器负责。主服务器遇到数据更新请求后会更改在内存中维护的管理数据，通过改造的 Paxos 协议通知其他备份服务器对相应的数据进行更新操作，并保证在多副本环境下的数据一致性；当多数备份服务器确认更新完成后，主服务器可以认为本次更新操作正确完成。其他所有备份服务器只是同步管理数据到本地，保持数据和主服务器完全一致。通信协议如图 3-4 所示。

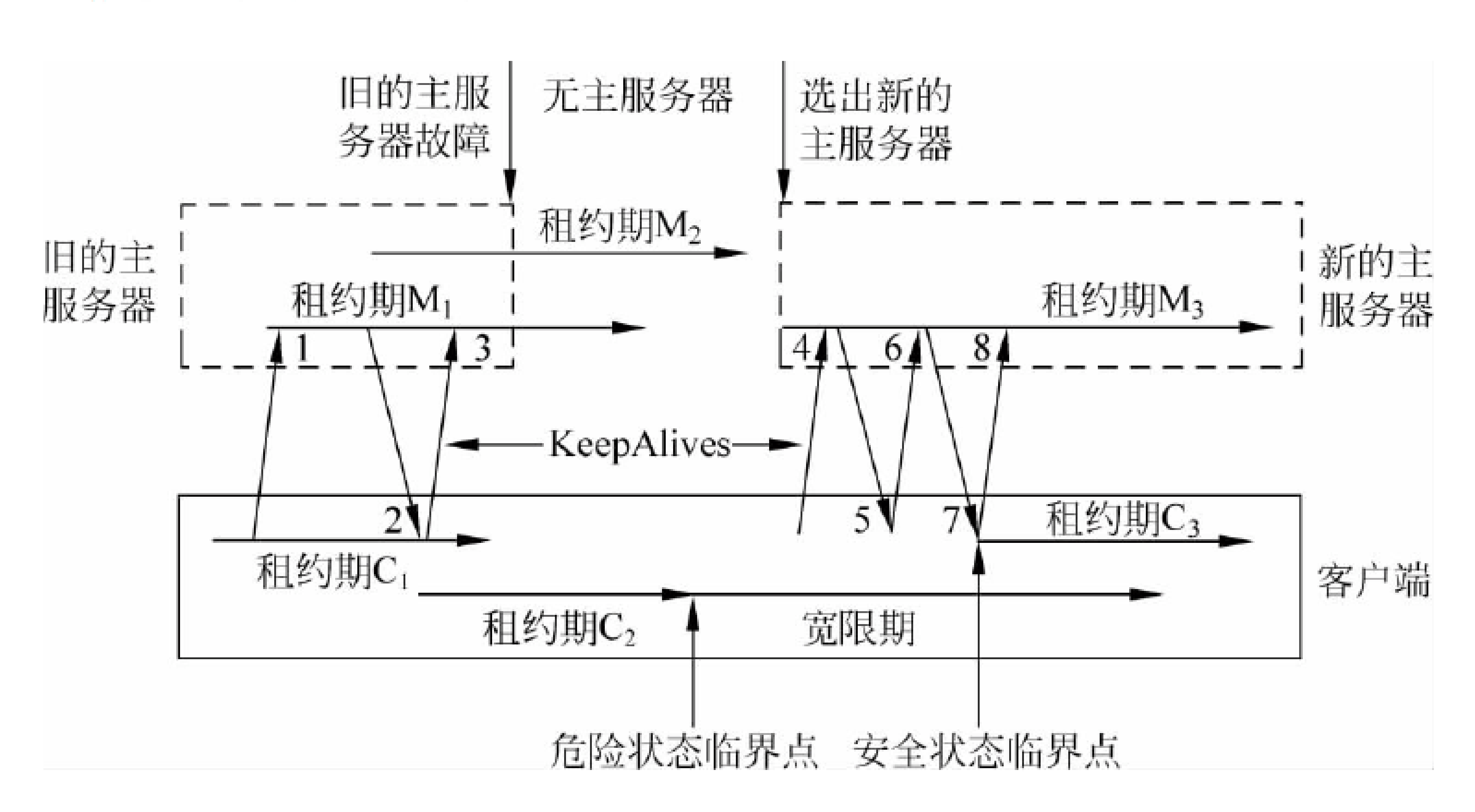


图 3-4　Client 与 Chubby 的通信

KeepAlive 是周期性发送的一种消息，它有两方面的功能：延长租约有效期，携带事件信息告诉客户端更新。事件包括文件内容的修改、子节点的增删改、Master 出错等。

在正常情况下，租约会由 KeepAlive 一直不断延长。如果 C_1 在未用完租约期时发现还需使用，便发送锁请求给 Master，Master 给它 Lease-M_1；C_2 在过了租约期后，发送锁请求给 Master，可是未收到 Master 的回答。其实此刻 Master 已经挂了，于是 Chubby 进入宽限期，在这期间 Chubby 要选举出新的 Master。Google 论文里对于这段时期有一个更形象的名字——Grace Period。在选举出 Master 后，新的主服务器下令前主服务器发的 Lease 失效，大家必须申请一份新的。然后 C_2 获得了 Lease-M_2。C_3 又恢复到正常情况。在图 3-4 中 4、5、6、7、8 是通过 Paxos 算法选举 Master 的颤抖期。在此期间最有可能产生问题，Amazon 的分布式服务就曾因此宕机，导致很长时间服务不可用。

3.3.2 ZooKeeper

ZooKeeper 是 Yahoo！公司开发的一套开源高吞吐分布式协同系统，目前已经在各种 NoSQL 数据库及诸多开源软件中获得广泛使用。分布式应用中的各节点可以通过 ZooKeeper 这个第三方来确保双方的同步，比如一个节点是发送，另一个节点是接收，但发送节点需要确认接收节点成功收到这个消息，因而就可以通过与一个可靠的第三方交互来获取接收节点的消息接收状态。

ZooKeeper 也是由多台同构服务器构成的一个集群，共用信息存储在集群系统中。共用信息采用树形结构来存储，用户可以将其看作一个文件系统，只是这些文件是一直存放在内存中的，文件存储容量受到内存的限制。

既然 ZooKeeper 可以被看作一个文件系统，那么它就具有文件系统相应的功能，只是在 ZooKeeper 和文件系统中功能的叫法不同。ZooKeeper 提供创建节点、删除节点、创建子节点、获取节点内容等功能。

ZooKeeper 服务由若干台服务器构成，每台服务器内存中维护相同的树形数据结构。其中的一台通过 ZAB 原子广播协议选举作为主服务器，其他的作为从服务器。客户端可以通过 TCP 协议连接任意一台服务器，如果是读操作请求，任意一个服务器都可以直接响应请求；如果是写数据操作请求，则只能由主服务器来协调更新操作。Chubby 在这一点上与 ZooKeeper 不同，所有的读/写操作都由主服务器完成，从服务器只是用于提高整个协调系统的可用性。

在带来高吞吐量的同时，ZooKeeper 的这种做法也带来了潜在的问题：客户端可能读到过期的数据。因为即使主服务器已经更新了某个内存数据，但是 ZAB 协议还未能将其广播到从服务器。为了解决这一问题，在 ZooKeeper 的接口 API 函数中提供了 Sync 操作，应用可以根据需要在读数据前调用该操作，其含义是接收到 Sync 命令的从服务器从主服务器同步状态信息，保证两者完全一致。

3.3.3 阿里云女娲协同系统

女娲(Nuwa)系统为飞天提供高可用的协调服务(Coordination Service),是构建各类分布式应用的核心服务,它的作用是采用类似文件系统的树形命名空间来让分布式进程互相协同工作。例如,当集群变更导致特定的服务被迫改变物理运行位置时,如服务器或者网络故障、配置调整或者扩容时,借助女娲系统可以使其他程序快速定位到该服务新的接入点,从而保证了整个平台的高可靠性和高可用性。

女娲系统基于类 Paxos 协议,由多个女娲 Server 以类似文件系统的树形结构存储数据,提供高可用、高并发用户请求的处理能力。

女娲系统的目录表示一个包含文件的集合。与 UNIX 中的文件路径一样,女娲中的路径是以"/"分割的,根目录(Root entry)的名字是"/",所有目录的名字都是以"/"结尾的。与 UNIX 文件路径的不同之处在于:女娲系统中的所有文件或目录都必须使用从根目录开始的绝对路径。由于女娲系统的设计目的是提供协调服务,而不是存储大量数据,所以每个文件的内容(Value)的大小被限制在 1MB 以内。在女娲系统中,每个文件或目录都保存有创建者的信息。一旦某个路径被用户创建,其他用户就可以访问和修改这个路径的值(即文件内容或目录包含的文件名)。

女娲系统支持 Publish/Subscribe 模式,其中一个发布者、多个订阅者(One Publisher/Many Subscriber)的模式提供了基本的订阅功能;另外,还可用通过多个发布者、多个订阅者(Many Publisher/Many Subscriber)的模式提供分布式选举(Distributed Election)和分布式锁的功能。

另外一个使用女娲系统来实现负载均衡的例子:提供某一服务的多个节点,在服务启动的时候在女娲系统的同一目录下创建文件,例如 server1 创建文件 nuwa://cluster/myservice/server1,server2 在同一目录下创建 nuwa://cluster/myservice/server2。当客户端使用远程过程调用时首先列举女娲系统服务中 nuwa://cluster/myservice 目录下的文件,这样就可以获得 server1 和 server2,客户端随后可以从中选择一个节点发出自己的请求,从而实现负载均衡。

3.3.4 ZooKeeper 在 HDFS 高可用方案中的使用

HDFS 由 3 个模块构成,分别包括 Client、NameNode 和 DataNode。NameNode 负责管理所有的 DataNode 节点,保存 block 和 DataNode 之间的对应信息,Client 读取文件和写入文件都需要 NameNode 节点的参与,因此 NameNode 发挥着至关重要的作用。在

当前设计中，NameNode 是单节点方式，存在单点故障问题，即 NameNode 节点宕机之后 HDFS 无法再对外提供数据存储服务，需要设计一种 HDFS NameNode 节点的高可用方法。总体来讲，维护 HDFS 高可用基于以下两个目的：

(1) 在出现 NameNode 节点故障时 HDFS 仍然可以对外提供数据的读取和写入服务。

(2) HDFS 会出现版本的更新迭代，以保证 HDFS 在更新过程中仍然可以对外提供服务。

HDFS 为了实现上述目的，采用的方式是再提供一个额外的 NameNode 节点，以此达到 HDFS 的高可用目的。在使用过程中部署两个 NameNode 节点，一个 NameNode 节点为 Active 节点，另一个 NameNode 节点是 Standby 节点。在正常情况下，Active 的 NameNode 节点服务正常的请求，一旦出现 Active NameNode 节点故障，则 Standby NameNode 节点切换变成 Active 节点，然后这个新的 Active NameNode 继续提供 NameNode 的功能，使 HDFS 可以继续正常工作。但是为了保证上述过程正常运行，需要解决以下问题：

(1) Standby 如何知道 Active 节点出现故障无法正常服务，需要探测系统何时出现故障。

(2) 当出现 Active NameNode 节点故障时，多个 Standby NameNode 节点如何选择一个新的 Active NameNode 节点。

一种解决上述问题的 HDFS 高可用方法是采用 ZK Failover Controller 的方法，具体结构如图 3-5 所示。

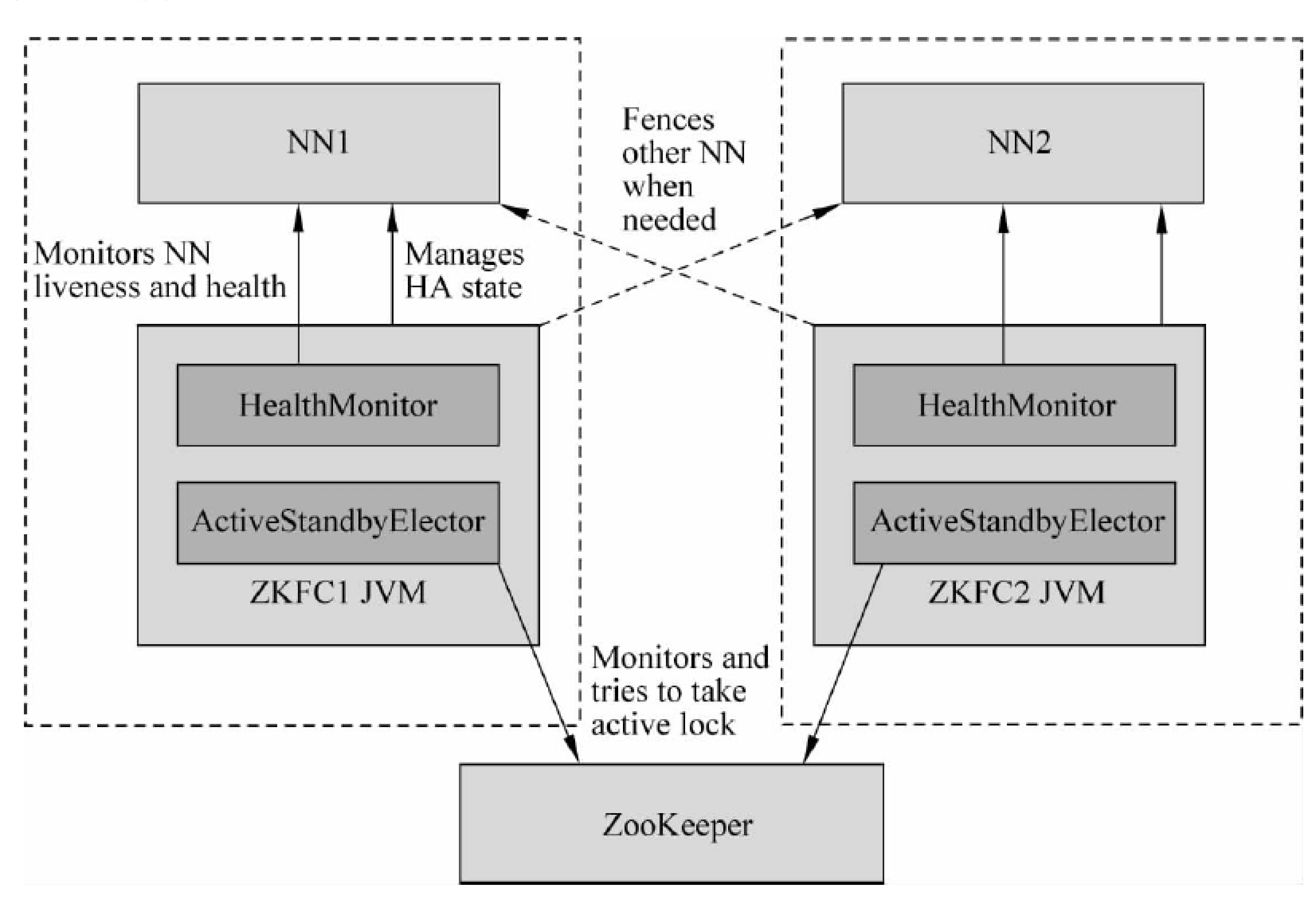


图 3-5 基于 ZooKeeper 的 HDFS 高可用方法

采用 ZK(ZooKeeper)设计 HDFS 高可用方案基于以下几点：

(1) ZooKeeper 提供了小规模的任意数据信息的强一致性。

(2) 可以在 ZooKeeper 集群中创建一个临时 znode 节点，当创建该 znode 节点的 Client 失效时，该临时 znode 节点会自动删除。

(3) 能够监控 ZooKeeper 集群中的一个 znode 节点的状态发生改变，并被异步通知。

上述设计的基于 ZK 的 HDFS 高可用方法由 ZKFC、HealthMonitor、ActiveStandby-Elector 几个主要部分组成。

(1) HealthMonitor 是一个线程，用于监控本地 NameNode 的状态信息，维持一个状态信息的视图，监控采用 RPC 方式。当状态信息发生改变时，通过 callback 接口方式发送消息给 ZKFC。

(2) ActiveStandbyElector 主要用于和 ZooKeeper 进行协调，ZKFC 与它通信主要由两个函数调用，分别是 joinElection 和 quitElection。

(3) ZKFailoverController 订阅来自 ActiveStandbyElector 和 HealthMonitor 的消息，同时管理 NameNode 的状态。

整体运行过程如下：启动的时候初始化 HealthMonitor 去监控本地 NameNode 节点，同时用 ZooKeeper 信息来初始化 ActiveStandbyElector，不立即把该 NameNode 节点加入选举。同时，随着 ActiveStandbyElector 和 HealthMonitor 状态的改变，ZKFC 做出对应的响应。

3.4 习题

1. 简述数据编码传输的好处。
2. 简要介绍 Snappy 压缩库，包括功能和数据格式。
3. 简要介绍 Chubby 的工作原理。
4. 简述 ZooKeeper 在 HDFS 高可用方案中发挥作用的理由。

第 4 章

大数据存储

随着结构化数据量和非结构化数据量的不断增长，以及分析数据来源的多样化，之前的存储系统设计已无法满足大数据应用的需求。对于大数据的存储，存在以下几个不容忽视的问题：

1. 容量

大数据时代存在的第一个问题就是“大容量”。“大容量”通常是指可达 PB 级的数据规模，因此海量数据存储系统的扩展能力也要得到相应等级的提升，同时其扩展还必须渐变，为此，通过增加磁盘柜或模块来增加存储容量，这样可以不需要停机。

2. 延迟

大数据应用不可避免地存在实时性的问题，大数据应用环境通常需要较高的 IOPS 性能。为了迎接这些挑战，小到简单的在服务器内用作高速缓存的产品，大到全固态介质可扩展存储系统，各种模式的固态存储设备应运而生。

3. 安全

大数据的分析往往需要对多种数据混合访问，这就催生出了一些新的、需要重新考虑的安全性问题。

4. 成本

成本控制是企业的关键问题之一，只有让每一台设备都实现更高的“效率”，才能控

制住成本。目前进入存储市场的重复数据删除、多数据类型处理等技术都可为大数据存储带来更大的价值，提升存储效率。

5. 灵活性

通常，大数据存储系统的基础设施规模都很大，为了保证存储系统的灵活性，使其能够随时扩容及扩展，必须经过详细的设计。

由于传统关系型数据库的局限性，传统的数据库已经不能很好地解决这些问题。在这种情况下，一些主要针对非结构化数据的管理系统开始出现。这些系统为了保障系统的可用性和并发性，通常采用多副本的方式进行数据存储。为了在保证低延时的用户响应时间的同时维持副本之间的一致状态，采用较弱的一致性模型，而且这些系统也普遍提供了良好的负载平衡策略和容错机制。

4.1　大数据存储技术的发展

在 20 世纪 50 年代中期以前，计算机主要用于科学计算，这个时候存储的数据规模不大，数据管理采用的是人工管理的方式；在 20 世纪 50 年代后期至 60 年代后期，为了更加方便管理和操作数据，出现了文件系统；从 20 世纪 60 年代后期开始，出现了大量的结构化数据，数据库技术蓬勃发展，开始出现了各种数据库，其中以关系型数据库备受人们喜爱。

在科学研究过程中，为了存储大量的科学计算，有 Beowulf 集群的并行文件系统 PVFS 做数据存储，在超级计算机上有 Lustre 并行文件系统存储大量数据，IBM 公司在分布式文件系统领域研制了 GPFS 分布式文件系统，这些都是针对高端计算采用的分布式存储系统。

进入到 21 世纪以后，互联网技术不断发展，其中以互联网为代表企业产生大量的数据，为了解决这些存储问题，互联网公司针对自己的业务需求和基于成本考虑开始设计自己的存储系统，典型代表是 Google 公司于 2003 年发表的论文 Google File System，其建立在廉价的机器上，提供了高可靠、容错的功能。为了适应 Google 的业务发展，Google 推出了 BigTable 这样一种 NoSQL 非关系型数据库系统，用于存储海量网页数据，数据存储格式为行、列簇、列、值的方式；与此同时亚马逊公司公布了他们开发的另外一种 NoSQL 系统——DynamoDB。后续大量的 NoSQL 系统不断涌现，为了满足互联网中的大规模网络数据的存储需求，其中 Facebook 结合 BigTable 和 DynamoDB 的优点，推出了 Cassandra 非关系型数据库系统。

开源社区对于大数据存储技术的发展更是贡献重大，其中包括底层的操作系统层面的存储技术，比如文件系统 btrfs 和 xfs 等。为了适应当前大数据技术的发展，支持高并发、多核以及动态扩展等，Linux 开源社区针对技术发展需求开发下一代操作系统的文件系统 btrfs，该文件系统在不断完善；同时也包括分布式系统存储技术，功不可没的是 apache 开源社区，其贡献和发展了 HDFS、HBase 等大数据存储系统。

总体来讲，结合公司的业务需求以及开源社区的蓬勃发展，当前大数据存储系统不断涌现。

4.2 海量数据存储的关键技术

大数据处理面临的首要问题是如何有效地存储规模巨大的数据。无论是从容量还是从数据传输速度，依靠集中式的物理服务器来保存数据是不现实的，即使存在这么一台设备可以存储所有的信息，用户在一台服务器上进行数据的索引查询也会使处理器变得不堪重负，因此分布式成为这种情况的很好的解决方案。要实现大数据的存储，需要使用几十台、几百台甚至更多的分布式服务器节点。为保证高可用、高可靠和经济性，海量数据多采用分布式存储的方式来存储数据，采用冗余存储的方式来保证存储数据的可靠性，即为同一份数据存储多个副本。

数据分片与数据复制的关系如图 4-1 所示。

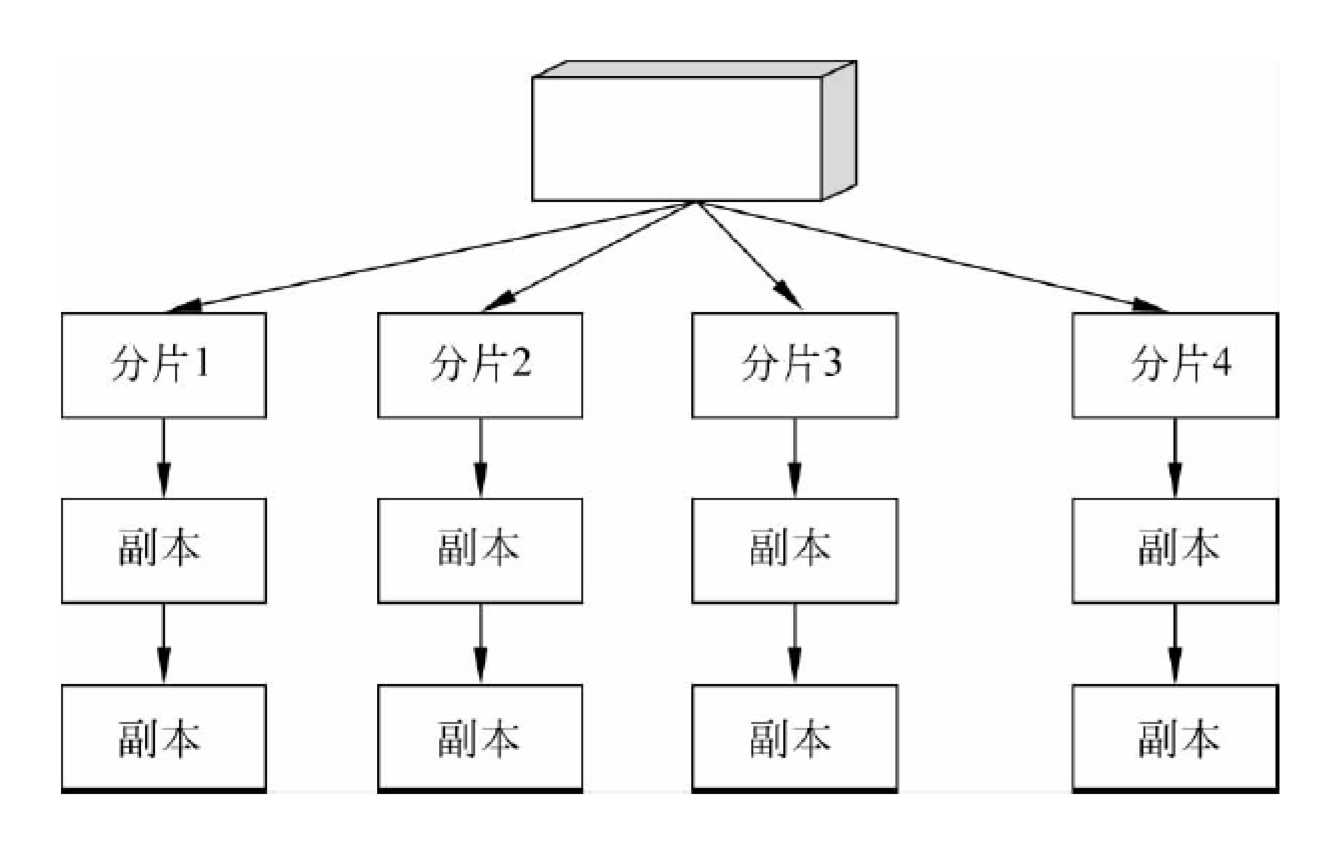


图 4-1 数据分片与数据复制

4.2.1 数据分片与路由

传统数据库采用纵向扩展方式，通过改善单机硬件资源配置来解决问题；主流大数据存储与计算系统采用横向扩展方式，支持系统可扩展性，即通过增加机器来获得水平

扩展能力。

对于海量数据，将数据进行切分并分配到各个机器中的过程叫分片（shard/partition），即将不同数据存放在不同节点。数据分片后，找到某条记录的存储位置称为数据路由（routing）。数据分片与路由的抽象模型如图 4-2 所示。

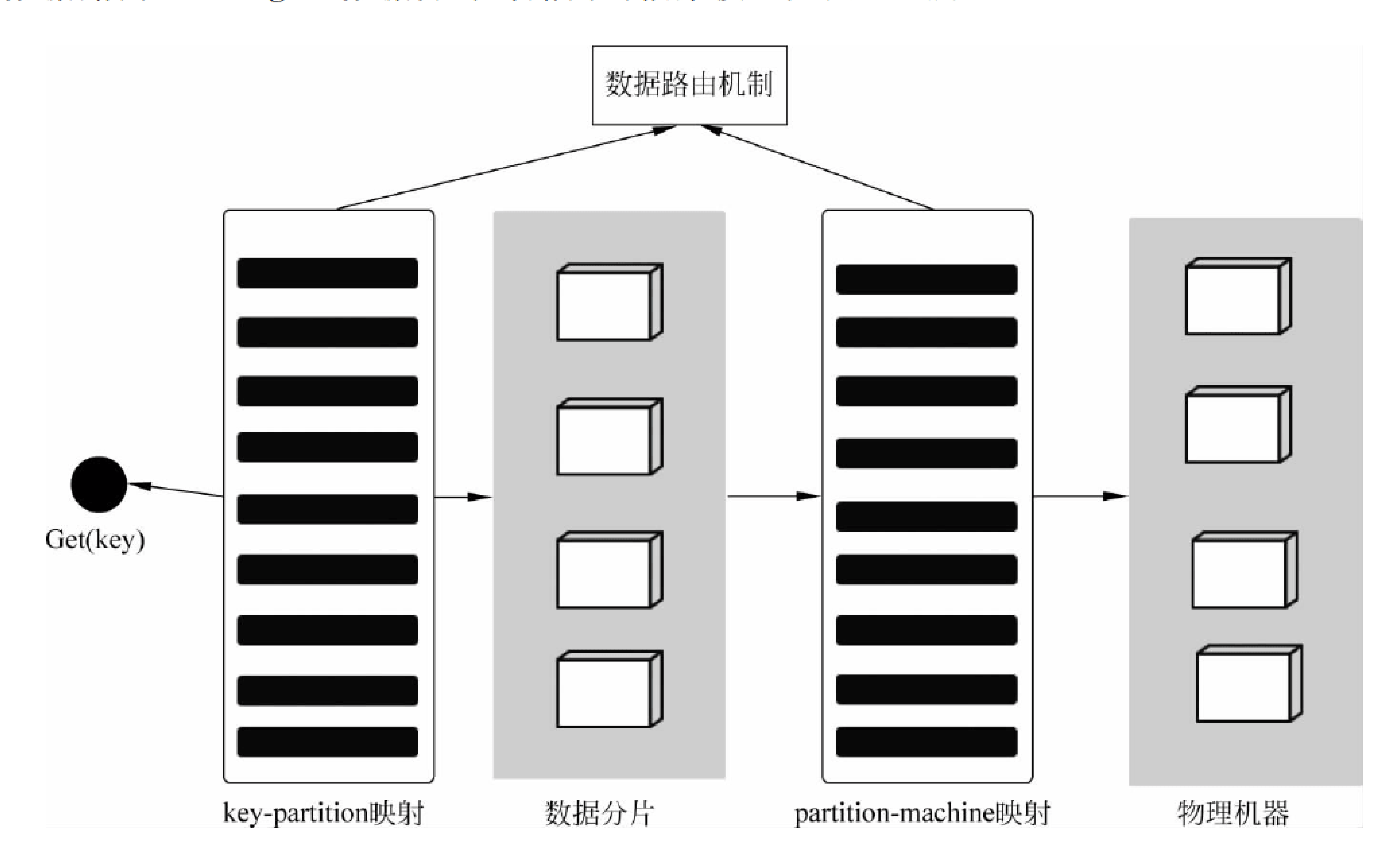


图 4-2 数据分片与路由的抽象模型

1. 数据分片

一般来说，数据库的繁忙体现在不同用户需要访问数据集中的不同部分。在这种情况下，把数据的各个部分存放在不同的服务器/节点中，每个服务器/节点负责自身数据的读取与写入操作，以此实现横向扩展，这种技术称为分片。

用户必须考虑以下两点。

（1）如何存放数据。可以实现用户从一个逻辑节点（实际多个物理节点的方式）获取数据，并且不用担心数据的存放位置。面向聚合的数据库可以很容易地解决这个问题。聚合结构是指把经常需要同时访问的数据存放在一起，因此可以把聚合作为分布数据的单元。

（2）如何保证负载平衡。即如何把聚合数据均匀地分布在各个节点中，让它们需要处理的负载量相等。负载分布情况可能会随着时间变化，因此需要一些领域特定的规则。比如有的需要按字典顺序，有的需要按逆域名序列等。

下面讲述一下分片类型。

1）哈希分片

采用哈希函数建立 Key-Partition 映射，其只支持点查询，不支持范围查询，主要有

Round Robin、虚拟桶、一致性哈希 3 种算法。

(1) Round Robin。其俗称哈希取模算法,这是实际中最常用的数据分片方法。若有 k 台机器,分片算法如下:

$$H(key)=hash(key) \bmod k$$

对物理机进行编号($0\sim k-1$),根据以上哈希函数,对于以 key 为主键的某个记录,H(key)的数值即是物理机在集群中的放置位置(编号)。

优点:实现简单。

缺点:缺乏灵活性,若有新机器加入,之前所有数据与机器之间的映射关系都被打乱,需要重新计算。

(2) 虚拟桶。在 Round Robin 的基础上,虚拟桶算法加入一个"虚拟桶层",形成两级映射。所有记录首先通过哈希函数映射到对应的虚拟桶(多对一映射)。虚拟桶和物理机之间再有一层映射(同样是多对一)。一般通过查找表来获知虚拟桶与物理机之间的映射关系。具体以 Membase 为例,如图 4-3 所示。

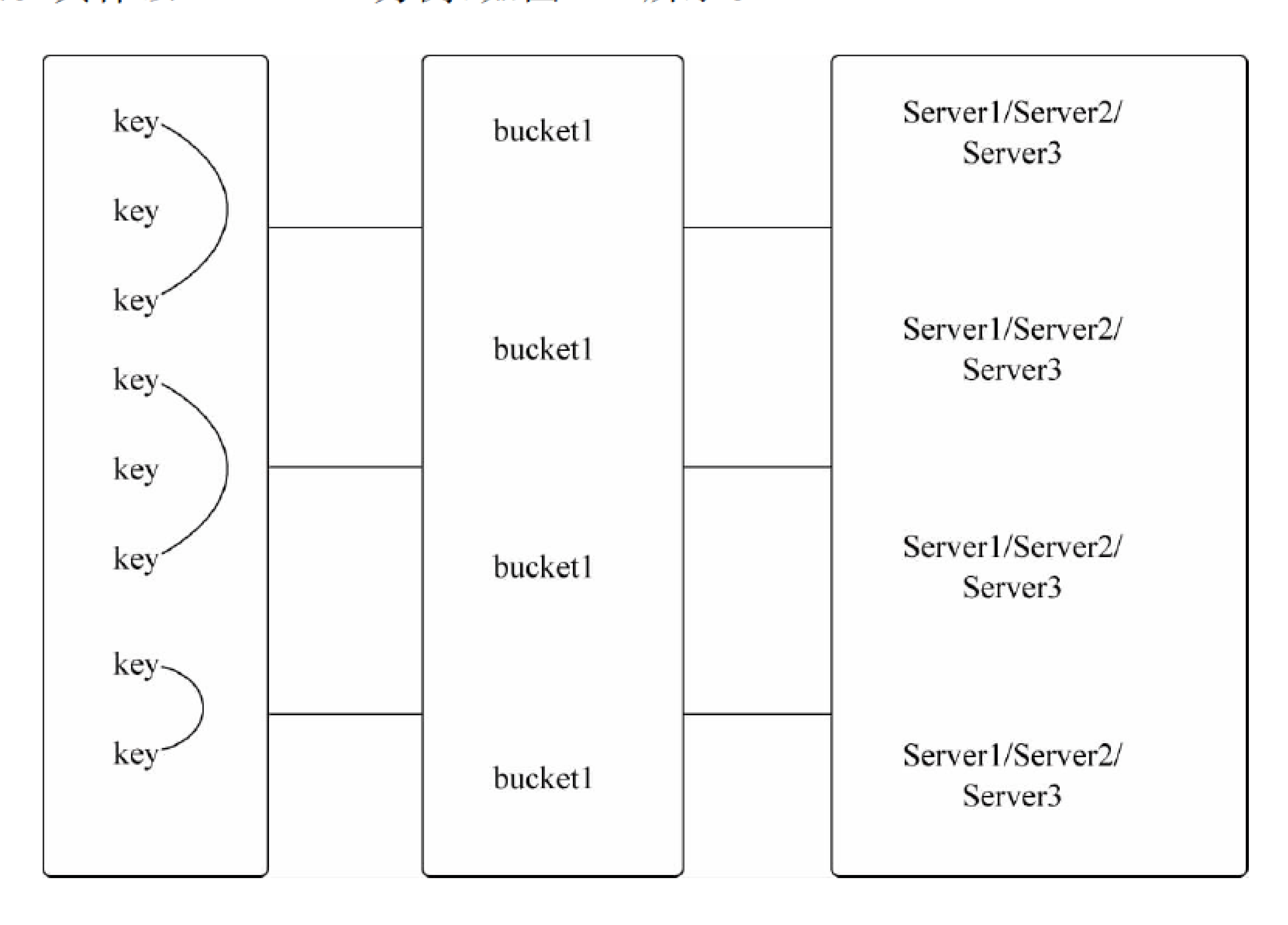


图 4-3 Membase 虚拟桶的运行

Membase 在待存储记录的物理机之间引入了虚拟桶层,所有记录首先通过哈希函数映射到对应的虚拟桶,记录和虚拟桶是多对一的关系,即一个虚拟桶包含多条记录信息;第二层映射是虚拟桶和物理机之间的映射关系,同样也是多对一映射,一个物理机可以容纳多个虚拟桶,具体是通过查找表来实现的,即 Membase 通过内存表管理这些映射关系。

对照抽象模型可以看出,Membase 的虚拟桶层对应数据分片层,一个虚拟桶就是一个数据分片。Key-Partition 映射采用映射函数。

与 Round Robin 相比，Membase 引入了虚拟桶层，这样将原先由记录直接到物理机的单层映射解耦成两级映射。当新加入机器时，将某些虚拟桶从原先分配的机器重新分配各机器，只需要修改 partition-machine 映射表中受影响的个别条目就能实现扩展。

优点：增加了系统扩展的灵活性。

缺点：实现相对麻烦。

（3）一致性哈希。一致性哈希是分布式哈希表的一种实现算法，将哈希数值空间按照大小组成一个首尾相接的环状序列，对于每台机器，可以根据 IP 和端口号经过哈希函数映射到哈希数值空间内。通过有向环顺序查找或路由表来查找。对于一致性哈希可能造成的各个节点负载不均衡的情况，可以采用虚拟节点的方式来解决。一个物理机节点虚拟成若干虚拟节点，映射到环状结构的不同位置。图 4-4 为哈希空间长度为 5 的二进制数值（$m=5$）的一致性哈希算法示意图。

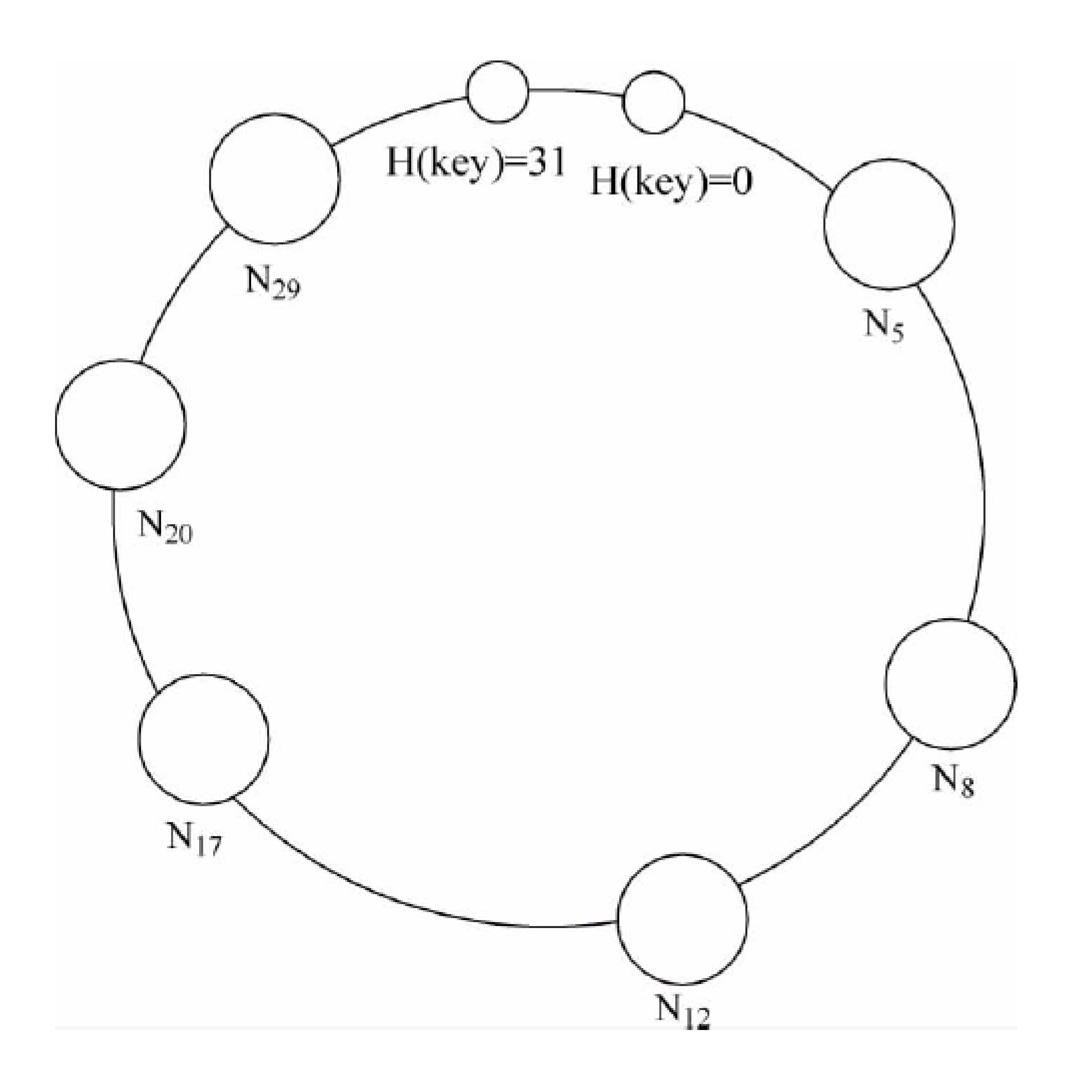


图 4-4　一致性哈希算法

在哈希空间可容纳长度为 32 的二进制数值（m＝32）空间里，每个机器根据 IP 地址或者端口号经过哈希函数映射到环内（图中 6 个大圆代表机器，后面的数字代表哈希值，即根据 IP 地址或者端口号经过哈希函数计算得出的在环状空间内的具体位置），而这台机器负责存储落在一段有序哈希空间内的数据，比如 N_{12} 节点存储哈希值在 9～12 范围内的数据，而 N_5 负责存储哈希值落在 30～31 和 0～5 范围内的数据。同时，每台机器还记录着自己的前驱和后继节点，成为一个真正意义上的有向环。

2）范围分片

范围分片首先将所有记录的主键进行排序，然后在排好序的主键空间里将记录划分成数据分片，每个数据分片存储有序的主键空间片段内的所有记录。

支持范围查询即给定记录主键的范围而一次读取多条记录，范围分片既支持点查

询，也支持范围查询。

分片可以极大地提高读取性能，但对于频繁写的应用帮助不大。同时，分片也可减少故障范围，只有访问故障节点的用户才会受影响，访问其他节点的用户不会受到故障节点的影响。

2. 路由

那么如何根据收到的请求找到储存的值呢？下面介绍 3 种方法。

1）直接查找法

如果哈希值落在自身管辖的范围内，则在此节点上查询，否则继续往后找，一直找到节点 N_x，x 是大于等于待查节点值的最小编号，这样一圈下来肯定能找到结果。

以图 4-4 为例，如有一个请求向 N_5 查询的主键为 H(key)＝6，因为此哈希值落在 N_5 和 N_8 之间，所以该请求的值存储在 N_8 的节点上，即如果哈希值落在自身管辖的范围内，则在此节点上查询，否则继续往后找，一直找到节点 N_x，x 是大于等于待查节点值的最小编号。

2）路由表法

直接查找法缺乏效率，为了加快查找速度，可以在每个机器节点配置路由表，路由表存储每个节点到每个除自身节点的距离，具体示例见表 4-1。

表 4-1 机器节点的路由表

距离	1	2	4	8	16
机器节点	N_{17}	N_{17}	N_{17}	N_{20}	N_{29}

在表 4-1 中，第 3 项代表与 N_{12} 的节点距离为 4 的哈希值（12＋4＝16）落在 N_{17} 节点身上，同理第 5 项代表与 N_{12} 的距离为 16 的哈希值落在 N_{29} 身上，这样找起来非常快速。

3）一致性哈希路由算法

同样如图 4-4，如请求节点 N_5 查询，N_5 的路由表如表 4-2 所示。

表 4-2 N_5 节点的路由表

距离	1	2	4	8	16
机器节点	N_8	N_8	N_{12}	N_{17}	N_{29}

假如请求的主键哈希值为 H(key)＝24，首先查询是否在 N_5 的后继节点上，发现后继节点 N_8 小于主键哈希值，则根据 N_5 的路由表查询，发现大于 24 的最小节点为 N_{29}（只有 29，因为 5＋16＝21＜24），因此哈希值落在 N_{29} 上。

4.2.2　数据复制与一致性

将同一份数据放置到多个节点（主从 master-slave 方式、对等式 peer-to-peer）的过程称为复制，数据复制可以保证数据的高可用性。

1. 主从复制

master-slave 模式，其中有一个 master 节点，存放重要数据，通常负责数据的更新，其余节点都叫 slave 节点，复制操作就是让 slave 节点的数据与 master 节点的数据同步。

优点：

（1）在频繁读取的情况下有助于提升数据的访问（读取 slave 节点分担压力），还可以增加多个 slave 节点进行水平扩展，同时处理更多的读取请求。

（2）可以增强读取操作的故障恢复能力。一个 slave 出故障，还有其他 slave 保证访问的正常进行。

缺点：数据一致性，如果数据更新没有通知到全部的 slave 节点，则会导致数据不一致。

2. 对等复制

主从复制有助于增强读取操作的故障恢复能力，对写操作频繁的应用没有帮助。它所提供的故障恢复能力只有在 slave 节点出错时才能体现出来，master 仍然是系统的瓶颈。对等复制是指两个节点相互为各自的副本，没有主从的概念。

优点：丢失其中一个节点不影响整个数据库的访问。

缺点：因为同时接受写入请求，容易出现数据不一致问题。在实际使用中，通常只有一个节点接受写入请求，另一个 master 作为候补，只有当对等的 master 出故障时才会自动承担写操作请求。

3. 数据一致性

有一个存储系统，其底层是一个复杂的高可用、高可靠的分布式存储系统。一致性模型的定义如下：

（1）强一致。按照某一顺序串行执行存储对象的读/写操作，更新存储对象之后，后续访问总是读到最新值。假如进程 A 先更新了存储对象，存储系统保证后续 A、B、C 的读取操作都将返回最新值。

（2）弱一致性。更新存储对象之后，后续访问可能读不到最新值。假如进程A先更新了存储对象，存储系统不能保证后续A、B、C的读取操作能读取到最新值。从更新成功这一刻开始算起，到所有访问者都能读到修改后的对象为止，这段时间称为"不一致性窗口"，在该窗口内访问存储时无法保证一致性。

（3）最终一致性。最终一致性是弱一致性的特例，存储系统保证所有访问将最终读到对象的最新值。例如，进程A写一个存储对象，如果对象上后续没有更新操作，那么最终A、B、C的读取操作都会读取到A写入的值。"不一致性窗口"的大小依赖于交互延迟、系统的负载，以及副本个数等。

4.3 重要数据结构和算法

分布式存储系统中存储大量的数据，同时需要支持大量的上层读/写操作，为了实现高吞吐量，设计和实现一个良好的数据结构能起到相当大的作用。典型的如LSM树结构，为NoSQL系统对外提供高吞吐量提供了更大的可能。在大规模分布式系统中需要查找到具体的数据，设计一个良好的数据结构，以支持快速的数据查找，如MemC3中的Cuckoo Hash，为MemC3在读多写少负载情况下极大地减少了访问延迟；HBase中的Bloom Filter结构，用于在海量数据中快速确定数据是否存在，减少了大量的数据访问操作，从而提高了总体的数据访问速度。

因此，一个良好的数据结构和算法对于分布式系统来说有着很大的作用。下面讲述当前大数据存储领域中一些比较重要的数据结构。

4.3.1 Bloom Filter

Bloom Filter用于在海量数据中快速查找给定的数据是否在某个集合内。

如果想判断一个元素是不是在一个集合内，一般想到的是将集合中的所有元素保存起来，然后通过比较确定，链表、树、散列表（又叫哈希表，Hash Table）等数据结构都是这种思路。但是随着集合中元素的增加，需要的存储空间越来越大，同时检索速度也越来越慢，上述3种结构的检索时间复杂度分别为$O(n)$、$O(\log n)$、$O(n/k)$。

Bloom Filter的原理是当一个元素被加入集合时，通过k个散列函数将这个元素映射成一个位数组中的k个点，把它们置为1。检索时，用户只要看看这些点是不是都是1就（大约）知道集合中有没有它了：如果这些点有任何一个0，则被检元素一定不在；如果都是1，则被检元素很可能在。这就是Bloom Filter的基本思想。

Bloom Filter 的高效是有一定代价的：在判断一个元素是否属于某个集合时，有可能会把不属于这个集合的元素误认为属于这个集合。因此，Bloom Filter 不适合那些“零错误”的应用场合。在能容忍低错误率的应用场合下，Bloom Filter 通过极少的错误换取了存储空间的极大节省。

下面具体来看 Bloom Filter 是如何用位数组表示集合的。初始状态时如图 4-5 所示，Bloom Filter 是一个包含 m 位的位数组，每一位都置为 0。

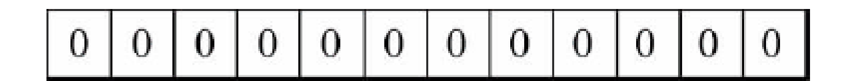


图 4-5 Bloom Filter 初始位数组

为了表达 $S=\{x_1, x_2, \cdots, x_n\}$这样一个 n 个元素的集合，Bloom Filter 使用 k 个相互独立的哈希函数(Hash Function)，它们分别将集合中的每个元素映射到$\{1,\cdots,m\}$的范围中。对任意一个元素 x，第 i 个哈希函数映射的位置 $h_i(x)$会被置为 1($1\leqslant i\leqslant k$)。注意，如果一个位置多次被置为 1，那么只有第一次会起作用，后面几次将没有任何效果。在图 4-6 中，$k=3$，且有两个哈希函数选中同一个位置(从左边数第 5 位，即第 2 个“1”处)。

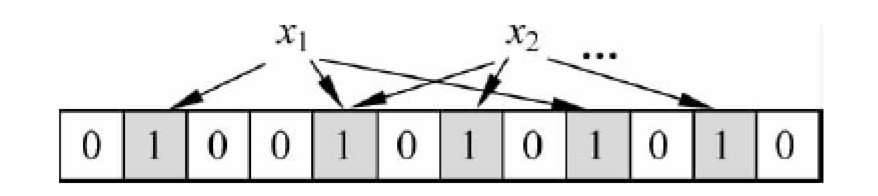


图 4-6 Bloom Filter 哈希函数

在判断 y 是否属于这个集合时，对 y 应用 k 次哈希函数，如果所有 $h_i(y)$的位置都是 1($1\leqslant i\leqslant k$)，那么就认为 y 是集合中的元素，否则就认为 y 不是集合中的元素。图 4-7 中的 y_1就不是集合中的元素(因为 y_1 有一处指向了 0 位)。y_2或者属于这个集合，或者不属于这个集合，如图 4-7 所示。

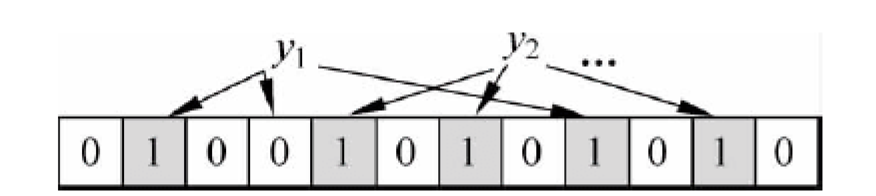


图 4-7 Bloom Filter 查找

这里举一个例子。有 A、B 两个文件，各存放 50 亿条 URL，每条 URL 占用 64 字节，内存限制是 4GB，试找出 A、B 文件共同的 URL。如果是 3 个乃至 n 个文件呢?

根据这个问题来计算一下内存的占用，4GB=2^{32}，大概是 43 亿，乘以 8 大概是 340 亿 bit，n=50 亿，如果按出错率 0.01 算大概需要 650 亿 bit。现在可用的是 340 亿，相差并不多，这样可能会使出错率上升一些。另外，如果这些 URL 和 IP 是一一对应的，就可以转换成 ip，这样就大大简单了。

4.3.2 LSM Tree

存储引擎和B树存储引擎一样，同样支持增、删、读、改、顺序扫描操作，而且可通过批量存储技术规避磁盘随机写入问题。但是LSM树和B+树相比，LSM树牺牲了部分读性能，用来大幅度提高写性能。

LSM树的原理是把一棵大树拆分成n棵小树，它首先写入内存中，随着小树越来越大，内存中的小树会fflush到磁盘中，磁盘中的树定期可以做merge操作，合并成一棵大树，以优化读性能。

对于最简单的二层LSM树而言，内存中的数据和磁盘中的数据做merge操作如图4-8所示。

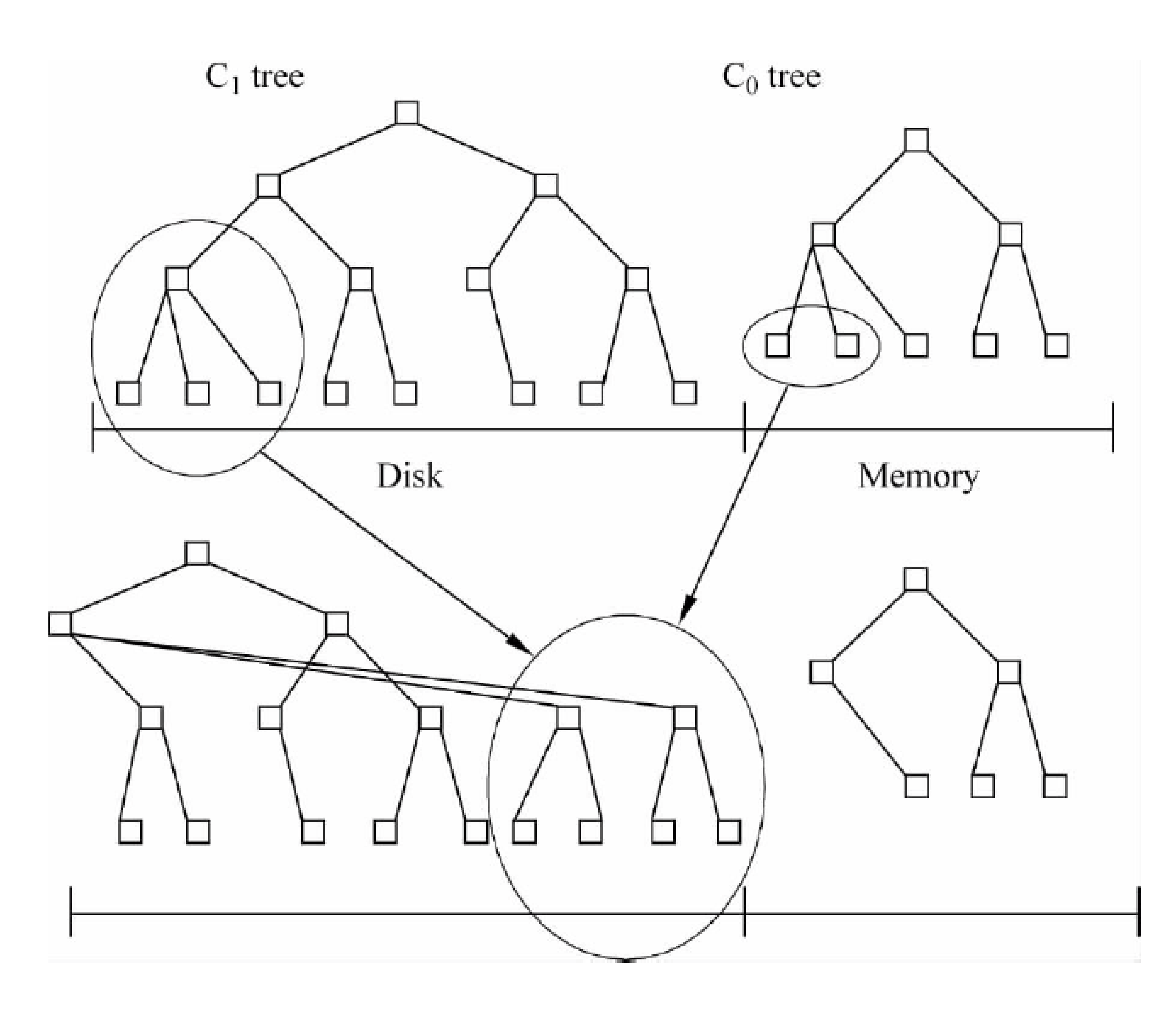


图4-8　LSM树

之前存在于磁盘的叶子节点被合并后，旧的数据并不会被删除，这些数据会复制一份和内存中的数据一起顺序写到磁盘。这样操作会有一些空间的浪费，但是LSM树提供了一些机制来回收这些空间。

磁盘中的树的非叶子节点数据也被缓存在内存中。

数据查找会首先查找内存中的树，如果没有查到结果，会转而查找磁盘中的树。

为什么LSM Tree的插入数据的速度比较快呢？

(1) 插入操作首先会作用于内存，由于内存中的树不会很大，因此速度快。

(2) 合并操作会顺序写入一个或多个磁盘页，比随机写入快得多。

4.3.3　Merkle Tree

Merkle Tree 是由计算机科学家 Ralph Merkle 提出的，并以他本人的名字来命名。本书将从数据“完整性校验”（检查数据是否有损坏）的角度介绍 Merkle Tree。

1. 哈希（Hash）

要实现完整性校验，最简单的方法就是对要校验的整个数据文件做哈希运算，将得到的哈希值发布在网上，当把数据下载后再次运算一下哈希值，如果运算结果相等，就表示下载过程中文件没有任何损坏。因为哈希的最大特点是，如果输入数据稍微变了一点，那么经过哈希运算，得到的哈希值将会变得完全不一样。构成的哈希拓扑结构如图 4-9 所示。

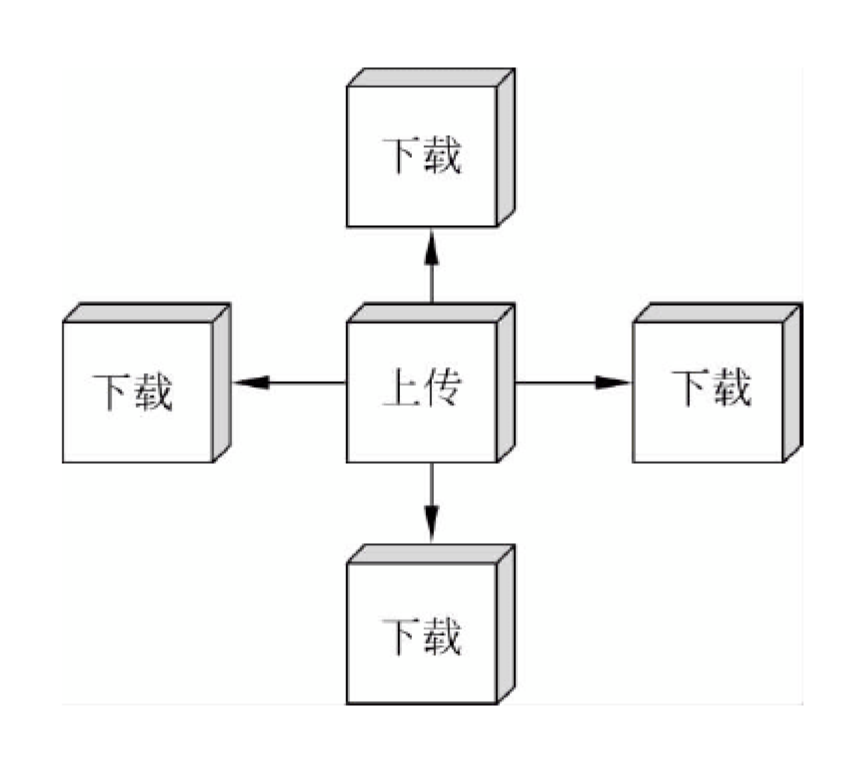


图 4-9　哈希拓扑

如果从一个稳定的服务器上进行下载，那么采用单个哈希进行校验的形式是可以接受的。

2. 哈希列表（Hash List）

但在点对点网络中进行数据传输时，如图 4-10 所示，我们会同时从多个机器上下载数据，而其中很多机器可以认为是不稳定或者是不可信的，这时需要有更加巧妙的做法。在实际中，点对点网络在传输数据的时候都是把比较大的一个文件切成小的数据块。这样的好处是如果有一小块数据在传输过程中损坏了，只要重新下载这一个数据块，不用重新下载整个文件。当然，这要求每个数据块都拥有自己的哈希值。在下载 BT 的时候，在下载真正的数据之前用户会先下载一个哈希列表。这时有一个问题出现了，如此多的哈希，我们怎么保证它们本身都是正确的呢？

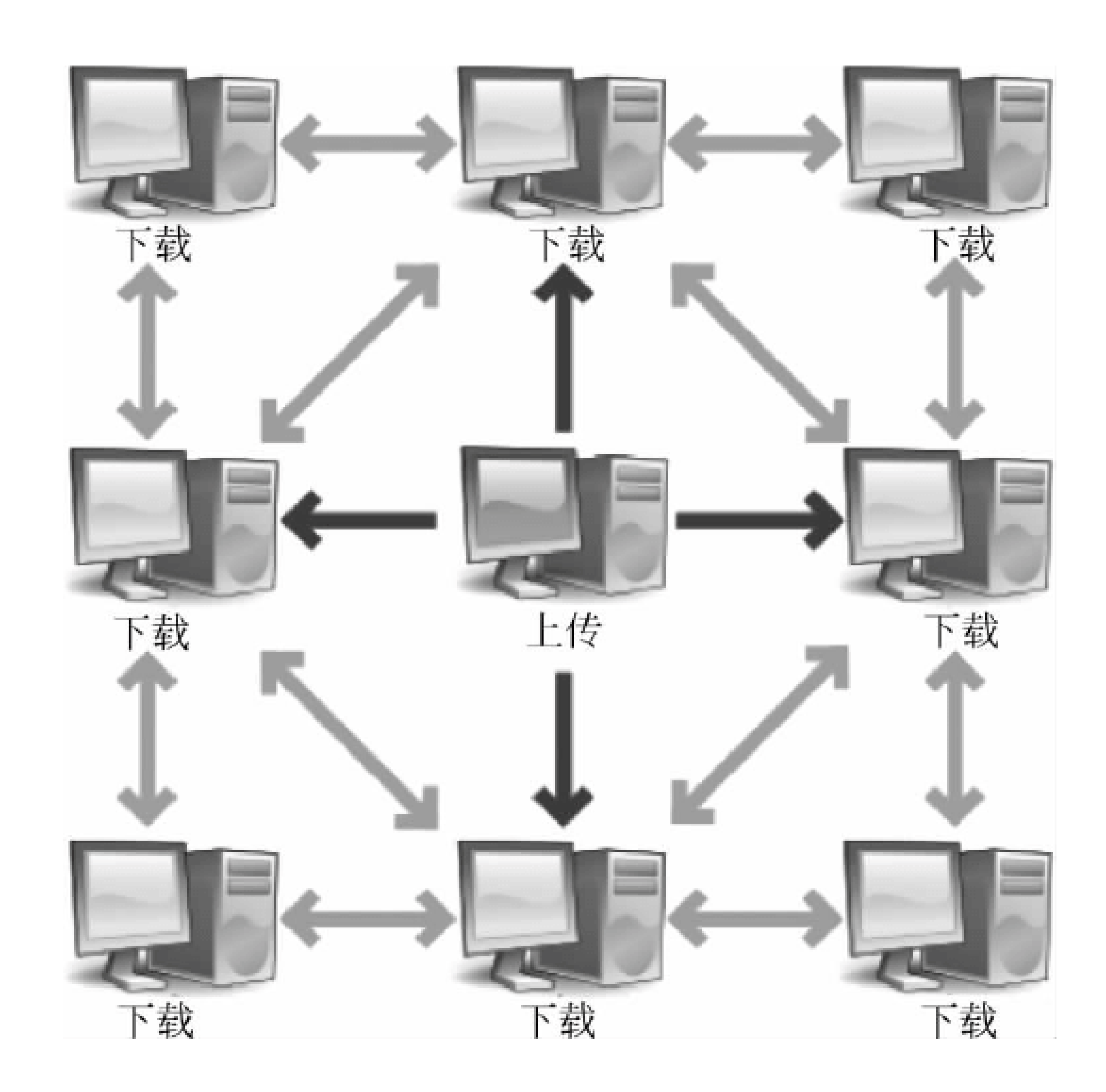


图 4-10 哈希列表

答案是我们需要一个根哈希，如图 4-11 所示，把每个小块的哈希值拼到一起，然后对这个长长的字符串再做一次哈希运算，最终的结果就是哈希列表的根哈希。如果我们能够保证从一个绝对可信的网站拿到一个正确的根哈希，就可以用它来校验哈希列表中的每一个哈希是否都是正确的，进而可以保证下载的每一个数据块的正确性。

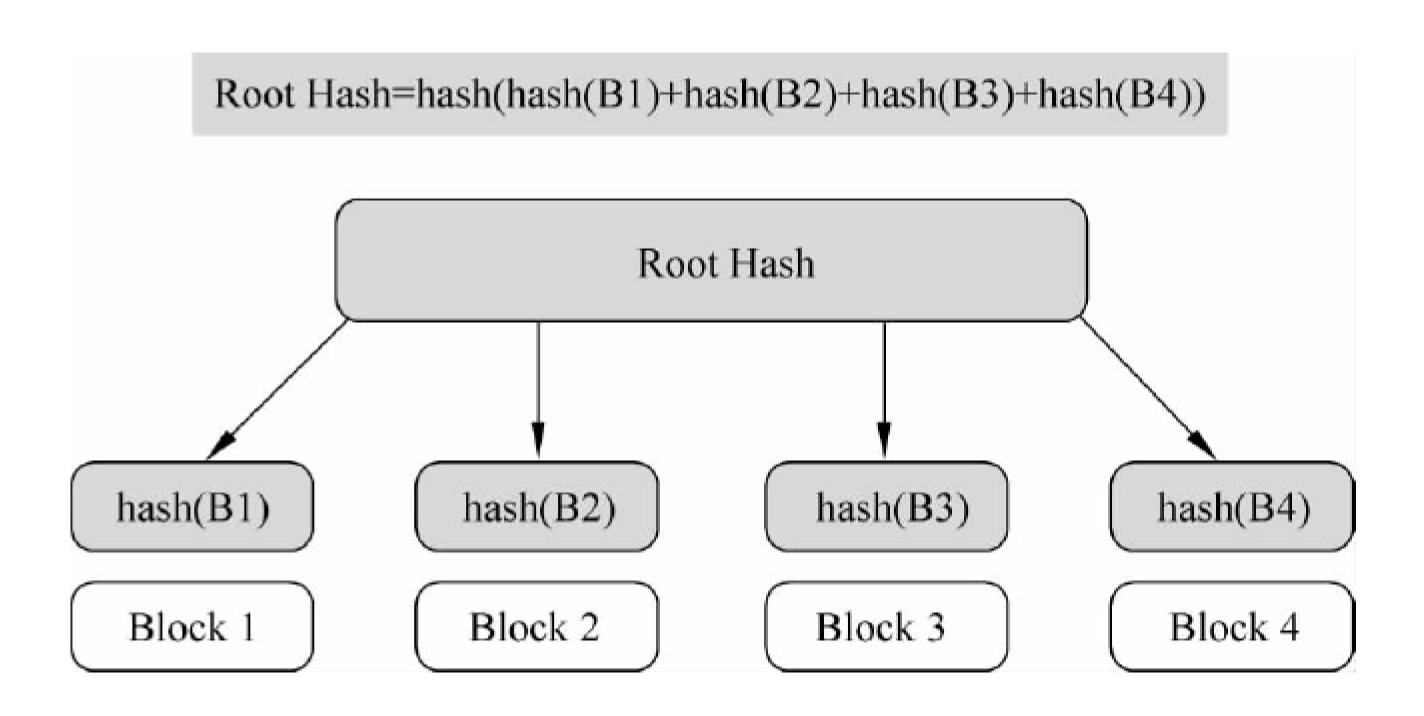


图 4-11 哈希流程

3. Merkle Tree 结构

在最底层，和哈希列表一样，我们把数据分成小的数据块，有相应的哈希和它对应。但是往上走，并不是直接运算根哈希，而是把相邻的两个哈希合并成一个字符串，然后运算这个字符串的哈希，这样每两个哈希组合得到了一个"子哈希"。如果最底层的哈希总数是单数，那么到最后必然出现一个单哈希，对于这种情况直接对它进行哈希运算，所以也能得到它的子哈希。于是往上推，依然是一样的方式，可以得到数目更少的新一级哈希，最终必然形成一棵倒着的树，到了树根的这个位置就剩下一个根哈希了，我们把它称

为 Merkle Root，如图 4-12 所示。

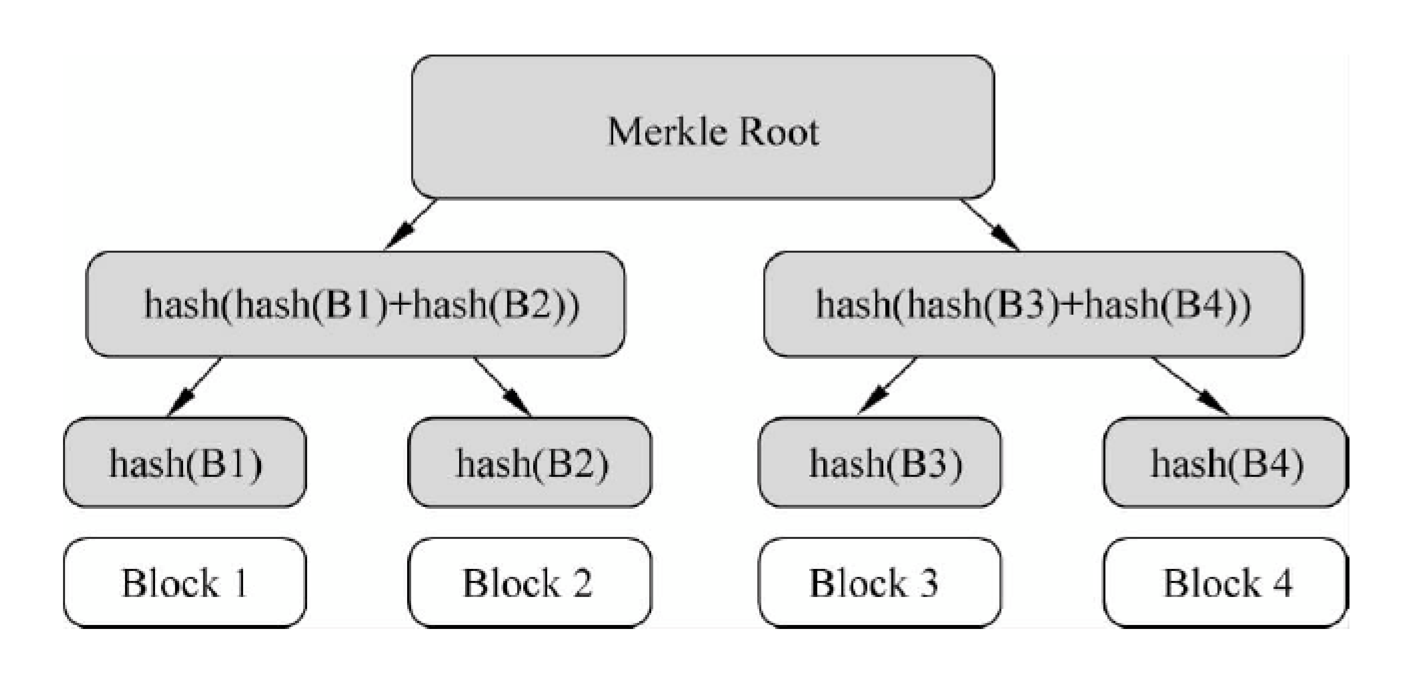


图 4-12　Merkle Tree 结构

相对于 Hash List，Merkle Tree 明显的一个好处是可以单独拿出一个分支来对部分数据进行校验，这是哈希列表所不能比拟的方便和高效。

4.3.4　Cuckoo Hash

Cuckoo 哈希是一种解决 hash 冲突的方法，其目的是使用简易的 hash 函数来提高 Hash Table 的利用率，保证 O(1)的查询时间也能够实现 hash key 的均匀分布。

基本思想是使用两个 hash 函数来处理碰撞，从而每个 key 都对应到两个位置。

插入操作如下：

(1) 对 key 值哈希，生成两个 hash key 值，hash k1 和 hash k2，如果对应的两个位置上有一个为空，直接把 key 插入即可。

(2) 否则，任选一个位置，把 key 值插入，把已经在那个位置的 key 值踢出。

(3) 被踢出来的 key 值需要重新插入，直到没有 key 被踢出为止。

其查找思路与一般哈希一致。

Cuckoo Hash 在读多写少的负载情况下能够快速实现数据的查找。

4.4　分布式文件系统

4.4.1　文件存储格式

文件系统最后都需要以一定的格式存储数据文件，常见的文件系统存储布局有行式存储、列式存储以及混合式存储 3 种，不同的类别各有其优缺点和适用的场景。在目前的大数据分析系统中，列式存储和混合式存储方案因其特殊优点被广泛采用。

1. 行式存储

在传统关系型数据库中，行式存储被主流关系型数据库广泛采用，HDFS 文件系统也采用行式存储。在行式存储中，每条记录的各个字段连续地存储在一起，而对于文件中的各个记录也是连续存储在数据块中，图 4-13 是 HDFS 的行式存储布局，每个数据块除了存储一些管理元数据外，每条记录都以行的方式进行数据压缩后连续存储在一起。

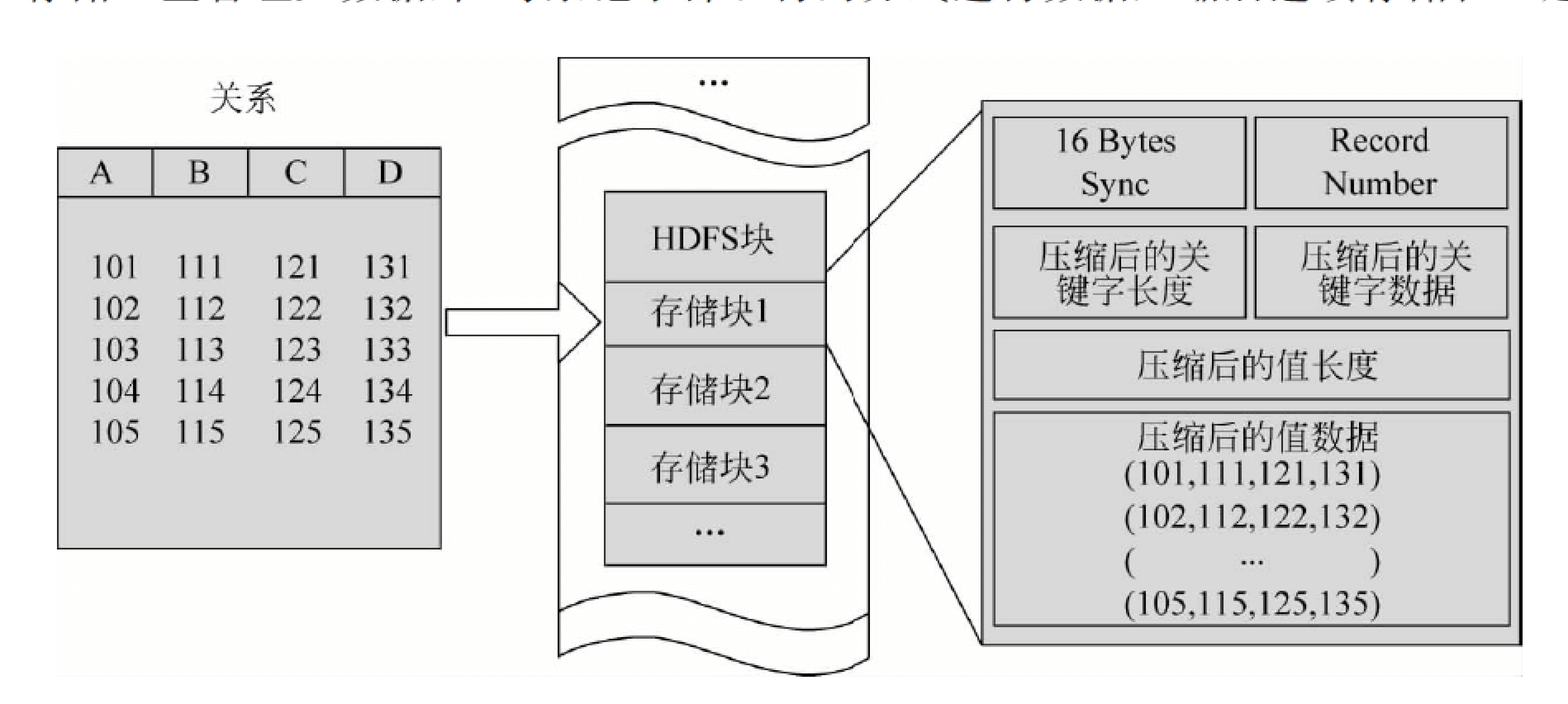


图 4-13　HDFS 的行式存储

行式存储对于大数据系统的需求已经不能很好地满足，主要体现在以下几个方面：

1）快速访问海量数据的能力被束缚

行的值由响应的列的值来定位，这种访问模型会影响快速访问的能力，因为在数据访问的过程中引入了耗时的输入/输出。在行式存储中，为了提高数据处理能力，一般通过分区技术来减少查询过程中数据输入/输出的次数，从而缩短响应时间。但是这种分区技术对海量数据规模下的性能改善效果并不明显。

2）扩展性差

在海量规模下，扩展性差是传统数据存储的一个致命的弱点。一般通过向上扩展(Scale up)和向外扩展(Scale out)来解决数据库扩展的问题。向上扩展是通过升级硬件来提升速度，从而缓解压力；向外扩展则是按照一定的规则将海量数据进行划分，再将原来集中存储的数据分散到不同的数据服务器上。但由于数据被表示成关系模型，从而难以被划分到不同的分片中等原因，这种解决方案仍然存在一定的局限性。

2. 列式存储

与行式存储布局对应，列式存储布局实际存储数据时按照列对所有记录进行垂直划分，将同一列的内容连续存放在一起。简单的记录数据格式类似于传统数据库的平面型数据结构，一般采取列组(Column Group/Column Family)的方式。典型的列式存储布局

是按照记录的不同列对数据表进行垂直划分,同一列的所有数据连续存储在一起,这样做有两个好处,一个好处是对于上层的大数据分析系统来说,如果查询操作只涉及记录的个别列,则只需读取对应的列内容即可,其他字段不需要进行读取操作;另一个好处是,因为数据按列存储,所以可以针对每列数据采取具有针对性的数据压缩算法,从而提升压缩率。但是列式存储的缺陷也很明显,对于 HDFS 这种按块存储的模式而言,有可能不同列分布在不同的数据块,所以为了拼合出完整的记录内容,可能需要大量的网络传输,导致效率低下。

采用列组方式存储布局可以在一定程度上缓解这个问题,也就是将记录的列进行分组,将经常使用的列分为一组,这样即使是按照列式来存储数据,也可以将经常联合使用的列存储在一个数据块中,避免通过不必要的网络传输来获取多列数据,对于某些场景而言会较大地提升系统性能。

在 HDFS 场景下,采用列组方式存储数据如图 4-14 所示,列被分为 3 组,A 和 B 分为一组,C 和 D 各自一组,即将列划分为 3 个列组并存储在不同的数据块中。

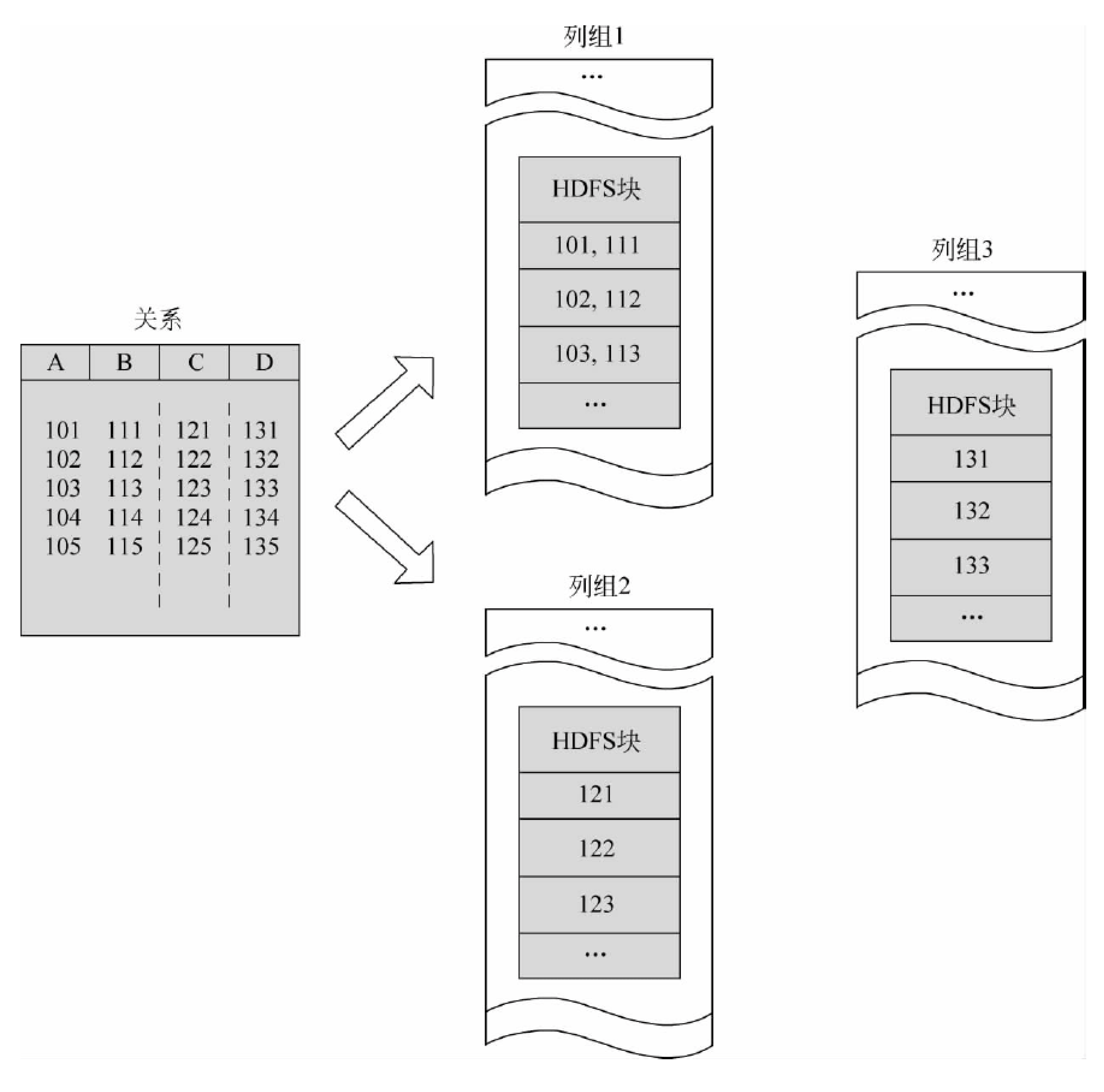


图 4-14　HDFS 列式存储布局

3. 混合式存储

尽管列式存储布局可以在一定程度上缓解上述的记录拼合问题，但是并不能彻底解决。混合式存储布局能够融合行式和列式存储布局的优点，能比较有效地解决这一问题。

混合式存储布局融合了行式和列式存储布局的优点，首先将记录表按照行进行分组，若干行划分为一组，而对于每组内的所有记录，在实际存储时按照列将同一列内容连续存储在一起。

4.4.2 GFS

GFS(Google File System)是 Google 公司为了存储百亿计的海量网页信息而专门开发的文件系统。在 Google 的整个大数据存储与处理技术框架中，GFS 是其他相关技术的基石，既提供了海量非结构化数据的存储平台，又提供了数据的冗余备份、成千台服务器的自动负载均衡以及失效服务器检测等各种完备的分布式存储功能。

考虑到 GFS 是在搜索引擎这个应用场景下开发的，在设计之初就定下了几个基本的设计原则。

首先，GFS 采用大量商业 PC 来构建存储集群。PC 的稳定性并没有很高的保障，尤其是大规模集群，每天都有机器宕机或者硬盘故障发生，这是 PC 集群的常态。因此，数据冗余备份、故障自动检测、故障机器自动恢复等都列在 GFS 的设计目标里。

其次，GFS 中存储的文件绝大多数是大文件，文件大小集中在 100MB 到几 GB 之间，所以系统设计应该对大文件的读/写操作做出有针对性的优化。

再次，系统中存在大量的“追加”写操作，即在已有文件的末尾追加内容，已经写入的内容不做更改；而很少有“随机”写行为，即在文件的某个特定位置之后写入数据。

最后，对于数据读取操作来说，绝大多数操作都是“顺序”读，少量的操作是“随机”读，即按照数据在文件中的顺序一次读入大量数据，而不是不断地在文件中定位到指定位置读取少量数据。

在下面的介绍中可以看到，GFS 的大部分技术思路都是围绕以上几个设计目标提出的。

在了解 GFS 整体架构之前首先了解一下 GFS 中的文件和文件系统。在应用开发者看来，GFS 文件系统类似于 Linux 文件系统中的目录和目录下的文件构成的树形结构。这个树形结构在 GFS 中被称为“GFS 命名空间”，同时，GFS 提供了文件的创建、删除、读取和写入等常见的操作接口。

上文说到，GFS 中大量存储的是大文件，文件大小超过几 GB 是很常见的。虽然文

件大小各异，但 GFS 在实际存储的时候首先将不同大小的文件切割成固定大小的数据块，每一个块称为一个“Chunk”。通常一个 Chunk 的大小设定为 64MB，这样每个文件就是由若干个固定大小的 Chunk 构成的。

GFS 以 Chunk 为基本存储单位，同一个文件的不同 Chunk 可能存储在不同的 ChunkServer 上，每个 ChunkServer 可以存储来自于不同文件的 Chunk。另外，在 ChunkServer 内部会对 Chunk 进一步切割，将其切割为更小的数据块，每一块被称为一个“Block”。Block 是文件读取的基本单位，即每次读取至少读一个 Block。

图 4-15 显示了 GFS 的整体架构，在这个架构中，主节点主要用来做管理工作，负责维护 GFS 命名空间和 Chunk 命名空间。在 GFS 系统内部，为了能识别不同的 Chunk，每个 Chunk 都被赋予一个唯一的编号，所有 Chunk 编号构成了 Chunk 命名空间。由于 GFS 文件被切割成了 Chunk，主节点还记录了每个 Chunk 存储在哪台 ChunkServer 上，以及文件和 Chunk 之间的映射关系。

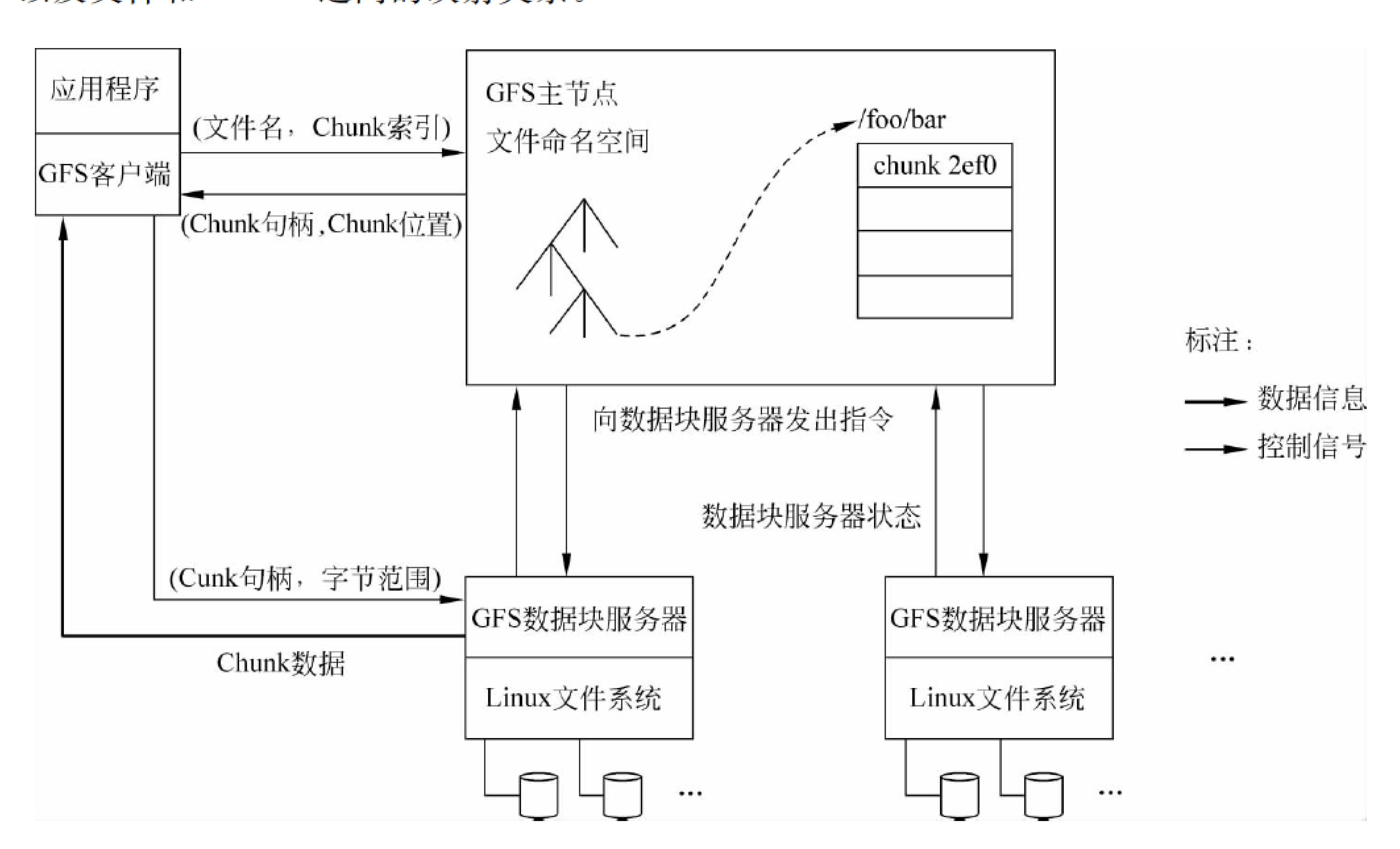


图 4-15　GFS 的整体架构

在 GFS 架构下，我们来看看“GFS 客户端”是如何读取数据的。

对于“GFS 客户端”来说，应用开发者提交的数据请求是从文件 file 中的位置 P 开始读取大小为 L 的数据。GFS 系统在收到这种请求后会在内部做转换，因为 Chunk 的大小是固定的，所以从位置 P 和大小 L 可以计算出要读的数据位于文件 file 的第几个 Chunk 中，请求被转换为< file，Chunk 序号>的形式。随后，这个请求被发送到 GFS 主节点，通过“主服务器”可以知道要读的数据在哪台 ChunkServer 上，同时可以将 Chunk

序号转换为系统内唯一的 Chunk 编号,并将这两个信息传回“GFS 客户端”。

“GFS 客户端”知道了应该去哪台 ChunkServer 读取数据后会和 ChunkServer 建立连接,并发送要读取的 Chunk 编号以及读取范围,ChunkServer 接收到请求后将请求的数据发送给“GFS 客户端”,如此就完成了一次数据读取的工作。

4.4.3 HDFS

Hadoop 分布式文件系统(HDFS)被设计成适合运行在商业硬件(commodity hardware)上的分布式文件系统。Hadoop 分布式文件系统和现有的分布式文件系统有很多共同点,但它和其他的分布式文件系统的区别也是很明显的。HDFS 是一个高度容错性的系统,适合部署在廉价的机器上。HDFS 能提供高吞吐量的数据访问,非常适合大规模数据集上的应用。HDFS 在最开始是作为 Apache Nutch 搜索引擎项目的基础架构开发的。HDFS 是 Apache Hadoop Core 项目的一部分。

HDFS 采用 master/slave 架构。一个 HDFS 集群由一个 namenode 和一定数目的 datanode 组成。namenode 是一个中心服务器,负责管理文件系统的名字空间(namespace)以及客户端对文件的访问。集群中的 datanode 一般是一个服务器,负责管理它所在节点上的存储。HDFS 呈现了文件系统的名字空间,用户能够以文件的形式在上面存储数据。从内部看,一个文件其实被分成一个或多个数据块,这些块存储在一组 datanode 上。namenode 执行文件系统的名字空间操作,比如打开、关闭、重命名文件或目录。它也负责确定数据块到具体 datanode 节点的映射。datanode 负责处理文件系统客户端的读/写请求。在 namenode 的统一调度下进行数据块的创建、删除和复制。HDFS 架构如图 4-16 所示。

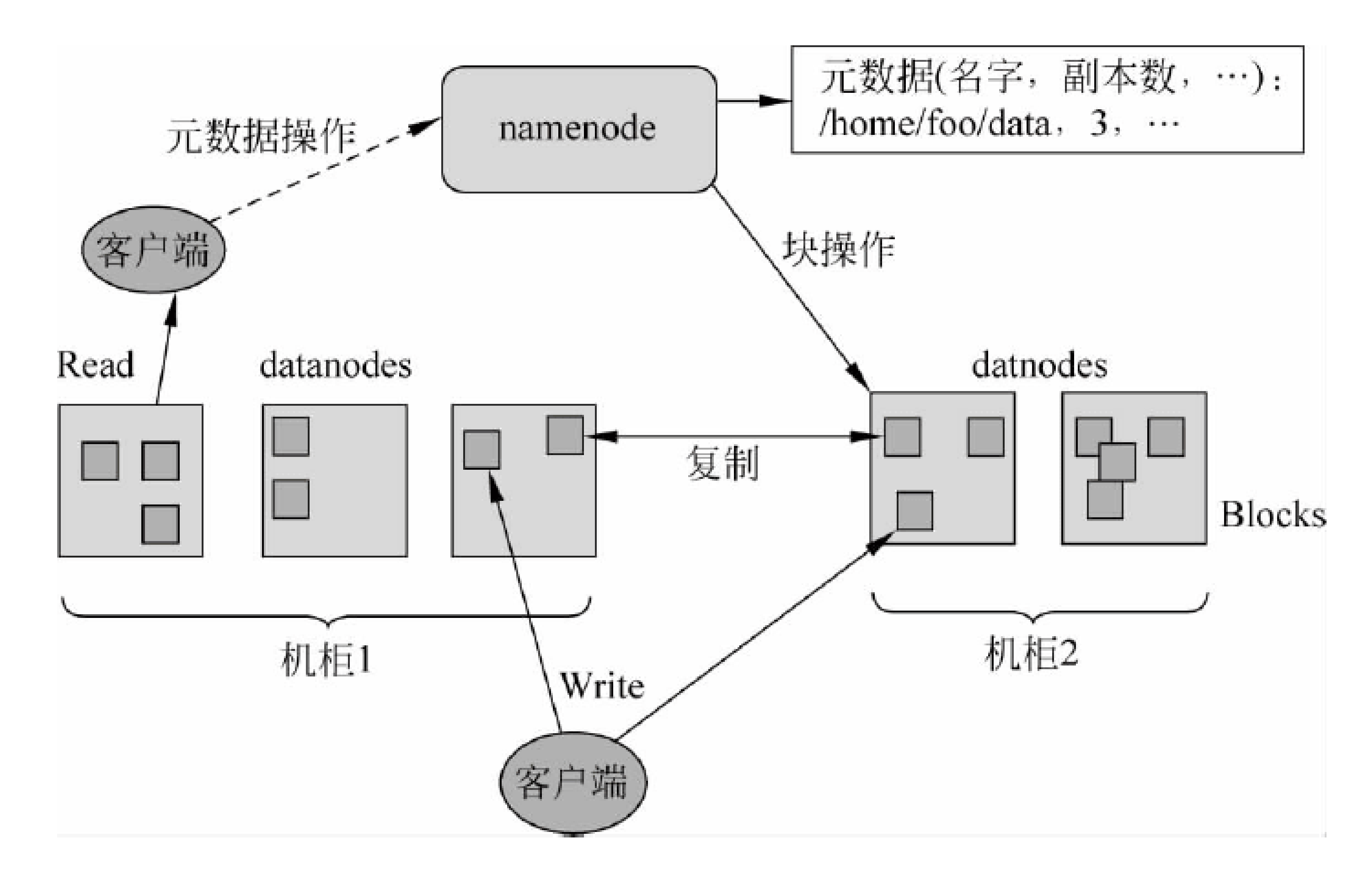


图 4-16 HDFS 架构

namenode 和 datanode 被设计成可以在普通的商用机器上运行，这些机器一般运行着 GNU/Linux 操作系统。

HDFS 采用 Java 语言开发，因此任何支持 Java 的机器都可以部署 namenode 或 datanode。由于采用了可移植性极强的 Java 语言，使得 HDFS 可以部署到多种类型的机器上。一个典型的部署场景是一台机器上只运行一个 namenode 实例，而集群中的其他机器分别运行一个 datanode 实例。这种架构并不排斥在一台机器上运行多个 datanode，但是这样的情况比较少见。

客户端访问 HDFS 中文件的流程如下：

(1) 从 namenode 获得组成这个文件的数据块位置列表。

(2) 根据位置列表得到储存数据块的 datanode。

(3) 访问 datanode 获取数据。

HDFS 保证数据存储可靠性的机理如下：

(1) 冗余副本策略。所有数据都有副本，对于副本的数目可以在 hdfs-site. xml 中设置相应的副本因子。

(2) 机架策略。采用一种"机架感知"相关策略，一般在本机架存放一个副本，在其他机架再存放别的副本，这样可以防止机架失效时丢失数据，也可以提高带宽利用率。

(3) 心跳机制。namenode 周期性地从 datanode 接受心跳信号和块报告，没有按时发送心跳的 datanode 会被标记为宕机，不会再给任何 I/O 请求，若是 datanode 失效造成副本数量下降，并且低于预先设置的阈值，namenode 会检测出这些数据块，并在合适的时机进行重新复制。

(4) 安全模式。namenode 启动时会先经过一个"安全模式"阶段。

(5) 校验和。客户端获取数据通过检查校验和发现数据块是否损坏，从而确定是否要读取副本。

(6) 回收站。删除文件会先到回收站，其里面的文件可以快速恢复。

(7) 元数据保护。映像文件和事务日志是 namenode 的核心数据，可以配置为拥有多个副本。

(8) 快照。支持存储某个时间点的映像，需要时可以使数据重返这个时间点的状态。

4.4.4　阿里云盘古

盘古系统是一个分布式文件系统，它是在整个阿里云计算"飞天"系统中负责数据存储的基石性系统，其上承载了一系列的云服务。盘古的设计目标是将大量通用机器的存储资源聚合在一起，为用户提供大规模、高可用、高吞吐量和良好扩展性的存储服务。

在整体架构上盘古采用 Master/ChunkServer 结构，Master 管理元数据，多 Master 之间采用 Primary-Secondary 模式，基于 Paxos 协议来保障服务的高可用；ChunkServer 负责实际数据的读/写，通过冗余副本提供数据安全；Client 对外提供类 POSIX 的专有 API，系统地提供丰富的文件形式，满足离线场景对高吞吐量的要求、在线场景下对低延迟的要求，以及虚拟机等特殊场景下随机访问的要求。其整体架构如图 4-17 所示。

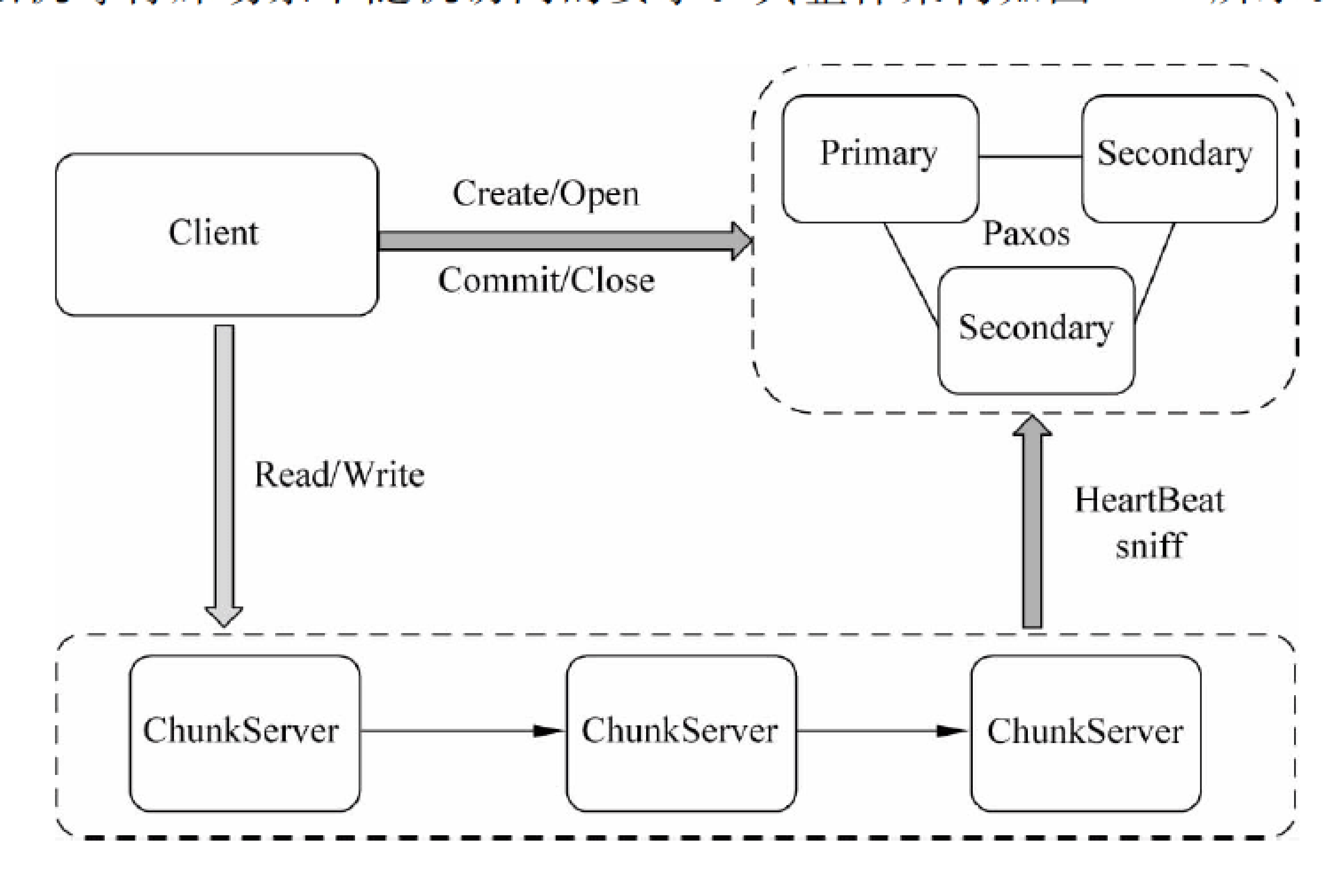


图 4-17　盘古分布式文件系统架构

整个盘古 Master 对外接口众多，根据是否需要在 Primary 和 Secondary 之间同步 Operation Log 分成读和写两大类。所有的读操作都不需要同步 Operation Log，所有的写操作基于数据一致性必须同步。考虑到 Primary 和 Secondary 随时可能发生切换，要保证数据一致，对于 Client 端发送的每一个写请求，Primary 必须在同步 Operation Log 完全成功后才能返回 Client。

4.5　分布式数据库 NoSQL

NoSQL 泛指非关系型数据库，相对于传统关系型数据库，NoSQL 有着更复杂的分类，包括 KV 数据库、文档数据库、列式数据库以及图数据库等。这些类型的数据库能够更好地适应复杂类型的海量数据存储。

4.5.1　NoSQL 数据库概述

一个 NoSQL 数据库提供了一种存储和检索数据的方法，该方法不同于传统的关系型数据库那种表格形式。NoSQL 形式的数据库从 20 世纪 60 年代后期开始出现，直到

21世纪早期，伴随着Web 2.0技术的不断发展，其中以互联网公司为代表，如Google、Amazon、Facebook等公司，带动了NoSQL这个名字的出现。目前NoSQL在大数据领域的应用非常广泛，应用于实时Web应用。

促进NoSQL发展的因素如下：

(1) 简单设计原则，可以更简单地水平扩展到多机器集群。

(2) 更细粒度地控制有效性。

一种NoSQL数据库的有效性取决于该类型NoSQL所能解决的问题。大多数NoSQL数据库系统都降低了系统的一致性，以利于有效性、分区容忍性和操作速度。当前制约NoSQL发展的很大部分是因为NoSQL的低级别查询语言、缺乏标准接口以及当前在关系型数据的投入。

目前大多数NoSQL提供了最终一致性，也就是数据库的更改最终会传递到所有节点上。表4-3是当前常用的NoSQL列表。

表4-3　常用的NoSQL列表

类　型	实　例
Key-Value Cache	Infinispan、Memcached、Repcached、Terracotta、Velocity
Key-Value Store	Flare、Keyspace、RAMCloud、SchemaFree、HyperDex、Aerospike
Data-Structures Server	Redis
Document Store	Clusterpoint、Couchbase、CouchDB、DocumentDB、Lotus Notes、MarkLogic、MongoDB
Object Database	DB4O、Objectivity/DB、Perst、Shoal、ZopeDB

4.5.2　KV数据库

KV数据库是最常见的NoSQL数据库形式，其优势是处理速度非常快，缺点是只能通过完全一致的键(Key)查询来获取数据。根据数据的保存形式，键值存储可以分为临时性和永久性，下面介绍两者兼具的KV数据库Redis。

Redis是著名的内存KV数据库，在工业界得到了广泛的使用。它不仅支持基本的数据类型，也支持列表、集合等复杂的数据结构，因此拥有较强的表达能力，同时又有非常高的读/写效率。Redis支持主从同步，数据可以从主服务器向任意数量的从服务器上同步，从服务器可以是关联其他从服务器的主服务器，这使得Redis可以执行单层树复制。由于完全实现了发布/订阅机制，使得从数据库在任何地方同步树时可订阅一个频道并接收主服务器完整的消息发布记录。同步对读取操作的可扩展性和数据冗余很有帮助。

对于内存数据库而言，最为关键的一点是如何保证数据的高可用性，应该说Redis在

发展过程中更强调系统的读/写性能和使用便捷性，在高可用性方面一直不太理想。

如图 4-18 所示，系统中有唯一的 Master（主设备）负责数据的读/写操作，可以有多个 Slave（从设备）来保存数据副本，数据副本只能读取不能更新。Slave 初次启动时从 Master 获取数据，在数据复制过程中 Master 是非阻塞的，即同时可以支持读/写操作。Master 采取快照结合增量的方式记录即时起新增的数据操作，在 Slave 就绪之后以命令流的形式传给 Slave，Slave 顺序执行命令流，这样就达到 Slave 和 Master 的数据同步。

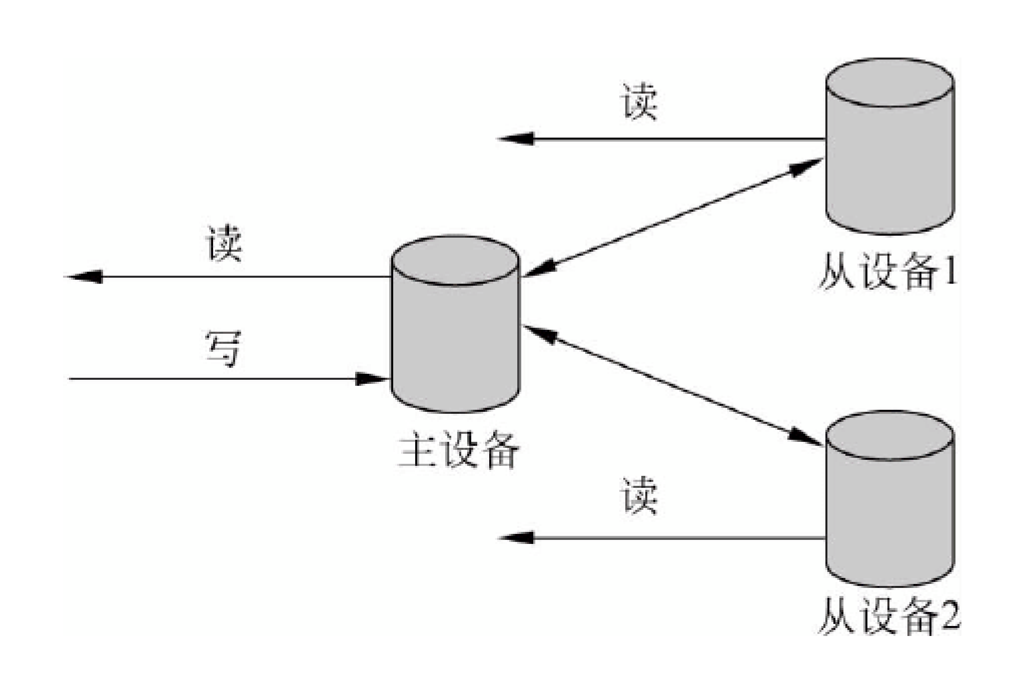


图 4-18　Redis 的副本维护策略

由于 Redis 采用这种异步的主从复制方式，所以 Master 接收到数据更新操作到 Slave 更新数据副本有一个时间差，如果 Master 发生故障可能导致数据丢失。而且 Redis 并未支持主从自动切换，如果 Master 故障，此时系统表现为只读，不能写入。由此可以看出 Redis 的数据可用性保障还是有缺陷的，那么在现版本下如何实现系统的高可用呢？一种常见的思路是使用 Keepalived 结合虚拟 IP 来实现 Redis 的 HA 方案。Keepalived 是软件路由系统，主要目的是为应用系统提供简洁强壮的负载均衡方案和通用的高可用方案。使用 Keepalived 实现 Redis 高可用方案如下：

首先在两台（或多台）服务器上分别安装 Redis 并设置主从。

其次，Keepalived 配置虚拟 IP 和两台 Redis 服务器的 IP 的映射关系，这样对外统一采用虚拟 IP，而虚拟 IP 和真实 IP 的映射关系及故障切换由 Keepalived 负责。当 Redis 服务器都正常时，数据请求由 Master 负责，Slave 只需要从 Master 复制数据；当 Master 发生故障时，Slave 接管数据请求并关闭主从复制功能，以避免 Master 再次启动后 Slave 数据被清掉；当 Master 恢复正常后，首先从 Slave 同步数据以获取最新的数据情况，关闭主从复制并恢复 Master 身份，与此同时 Slave 恢复其 Slave 身份。通过这种方法即可在一定程度上实现 Redis 的 HA。

4.5.3　列式数据库

列式数据库基于列式存储的文件存储格局，兼具 NoSQL 和传统数据库的一些优点，

具有很强的水平扩展能力、极强的容错性以及极高的数据承载能力，同时也有接近于传统关系型数据库的数据模型，在数据表达能力上强于简单的KV数据库。

下面以BigTable和HBase为例介绍列式数据库的功能和应用。

BigTable是Google公司设计的分布式数据存储系统，针对海量结构化或半结构化的数据，以GFS为基础，建立了数据的结构化解释，其数据模型与应用更贴近。目前BigTable已经在超过60个Google产品和项目中得到了应用，其中包括Google Analysis、Google Finance、Orkut和Google Earth等。

BigTable的数据模型本质上是一个三维映射表，其最基础的存储单元由行主键、列主键、时间构成的三维主键唯一确定。BigTable中的列主键包含两级，其中第一级被称为"列簇"(Column Families)，第二级被称为列限定符(Column Qualifier)，两者共同构成一个列的主键。

在BigTable内可以保留随着时间变化的不同版本的同一信息，这个不同版本由"时间戳"维度进行区分和表达。

HBase是一个开源的非关系型分布式数据库，它参考了Google的BigTable模型，实现的编程语言为Java。它是Apache软件基金会的Hadoop项目的一部分，运行于HDFS文件系统之上，为Hadoop提供类似于BigTable规模的服务。因此，它可以容错地存储海量稀疏的数据。HBase在列上实现了BigTable论文提到的压缩算法、内存操作和布隆过滤器BloomFilter。HBase的表能够作为MapReduce任务的输入和输出，可以通过Java API来访问数据，也可以通过REST、Avro或者Thrift的API来访问。HBase的整体架构如图4-19所示。

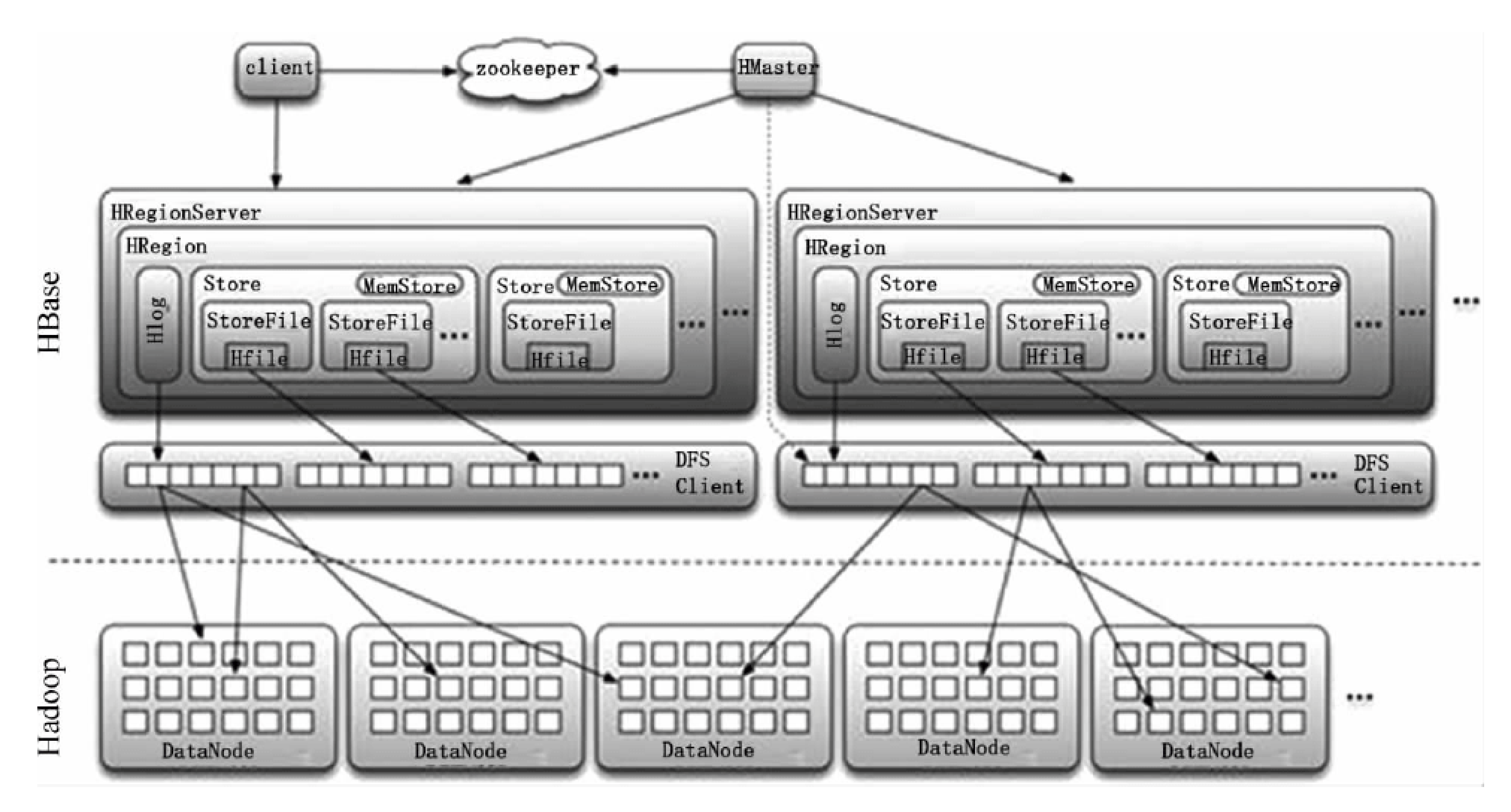


图4-19 HBase的架构图

HBase 以表的形式存放数据。表由行和列组成，每个列属于某个列簇，由行和列确定的存储单元称为元素，每个元素保存了同一份数据的多个版本，由时间戳来标识区分，如表 4-4 所示。

表 4-4　HBase 存储结构

行键	时间戳	列"contents:"	列"anchor:"		列"mine:"
"com.cnn.www"	t_9		"anchor:connsi.com"	"CNN"	
	t_8		"anchor:my.look.ca"	"CNN.com"	
	t_6	"< html >…"			"text/html"
	t_5	"< html >…"			
	t_3	"< html >…"			

4.5.4　图数据库

在图的领域并没有一套被广泛接受的术语，存在着很多不同类型的图模型。但是，有人致力于创建一种属性图形模型(Property Graph Model)，以期统一大多数不同的图实现。按照该模型，属性图里信息的建模使用下面 3 种构造单元：

- 节点(即顶点)；
- 关系(即边)，具有方向和类型(标记和标向)；
- 节点和关系上面的属性(即特性)。

更特殊的是，这个模型是一个被标记和标向的属性多重图(multigraph)。被标记的图的每条边都有一个标签，它被用来作为那条边的类型。有向图允许边有一个固定的方向，从末或源节点到首或目标节点。属性图允许每个节点和边有一组可变的属性列表，其中的属性是关联某个名字的值，简化了图形结构。多重图允许两个节点之间存在多条边。这意味着两个节点可以由不同边连接多次，即使两条边有相同的尾、头和标记。

图 4-20 是一个被标记的小型属性图。

下面以 Neo4j 这个具体的图数据库介绍图数据库的特性。Neo4j 是基于 Java 开发的开源图数据库，也是一种 NoSQL 数据库。Neo4j 在保证对数据关系的良好刻画的同时还支持传统关系型数据的 ACID 特性，并且在存储效率、集群支持以及失效备援等方面都有着不错的表现。

在所支持的数据类型上，Neo4j 支持两种数据类型，具体结构如图 4-21 所示。

(1) 节点。节点类似于 E-R 图中的实体(entity)，每个实体可以有 0 到多个属性，这些属性以 Key-Value 对的形式存在，并且对属性没有类别要求，也无须提前定义。另外，还允许给每个节点打上标签，以区别不同类型的节点。

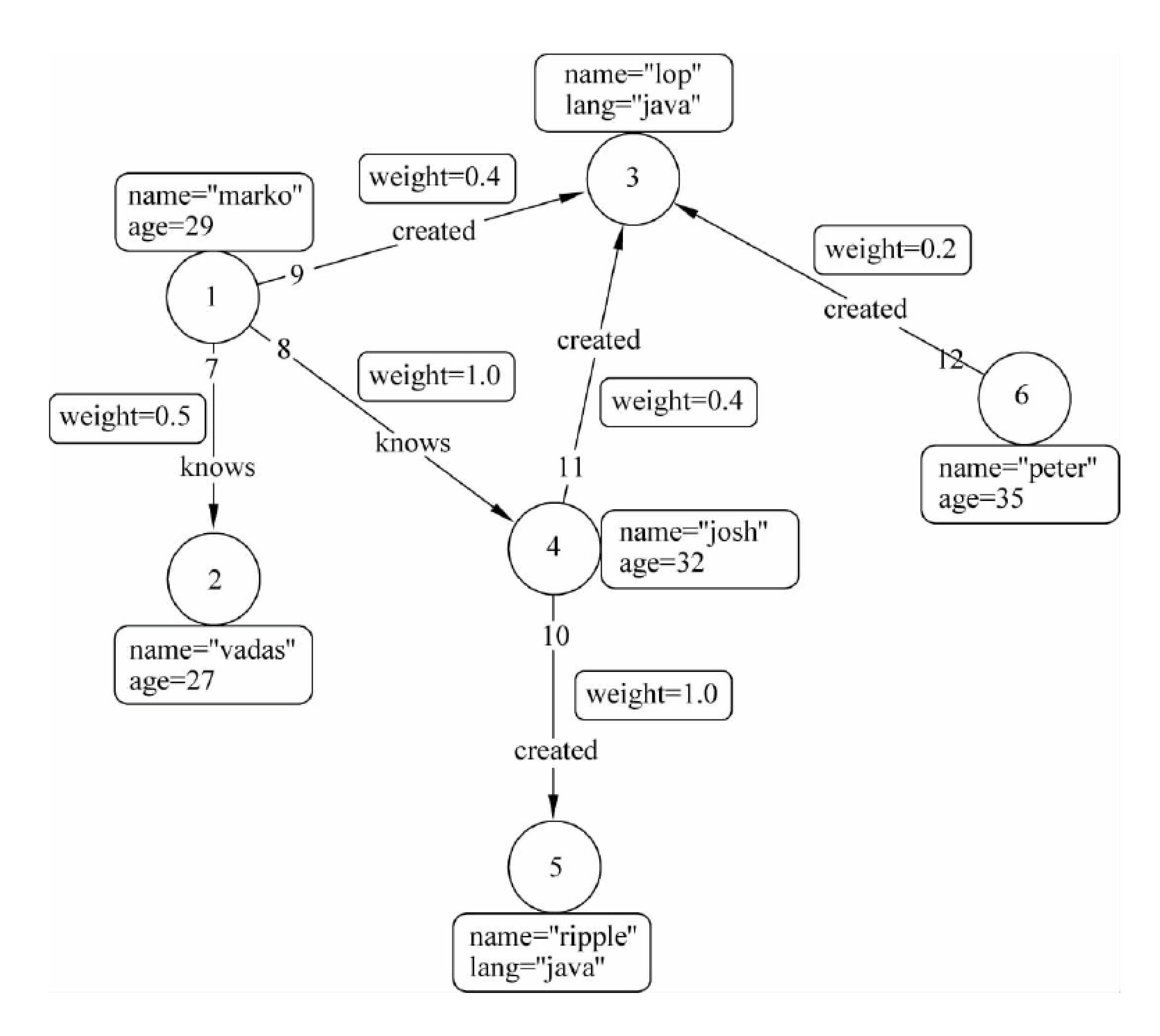


图 4-20　小型属性图

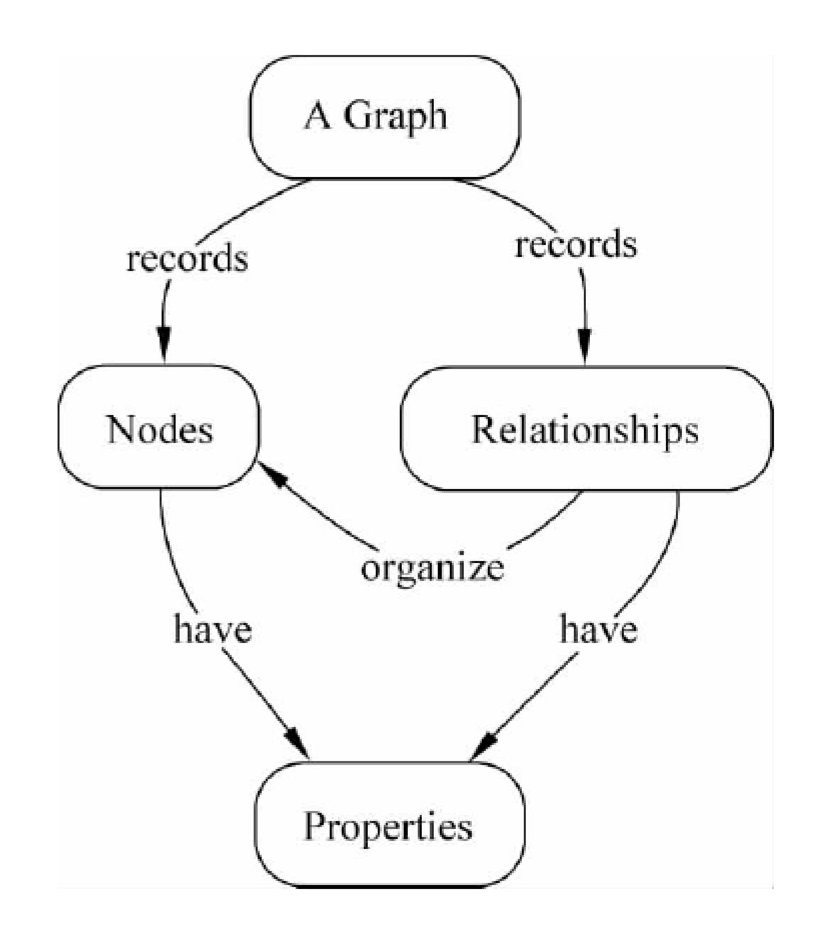


图 4-21　Neo4j 数据类型

(2) 关系。关系类似于 E-R 图中的关系(relationship),一个关系由一个起始节点和一个终止节点构成。另外和 node 一样,关系也可以有多个属性和标签。

一个实际的图数据库实例如图 4-22 所示。

Neo4j 具有以下特性:

(1) 关系在创建的时候就已经实现了,因而在查询关系的时候是一个 O(1)的操作。

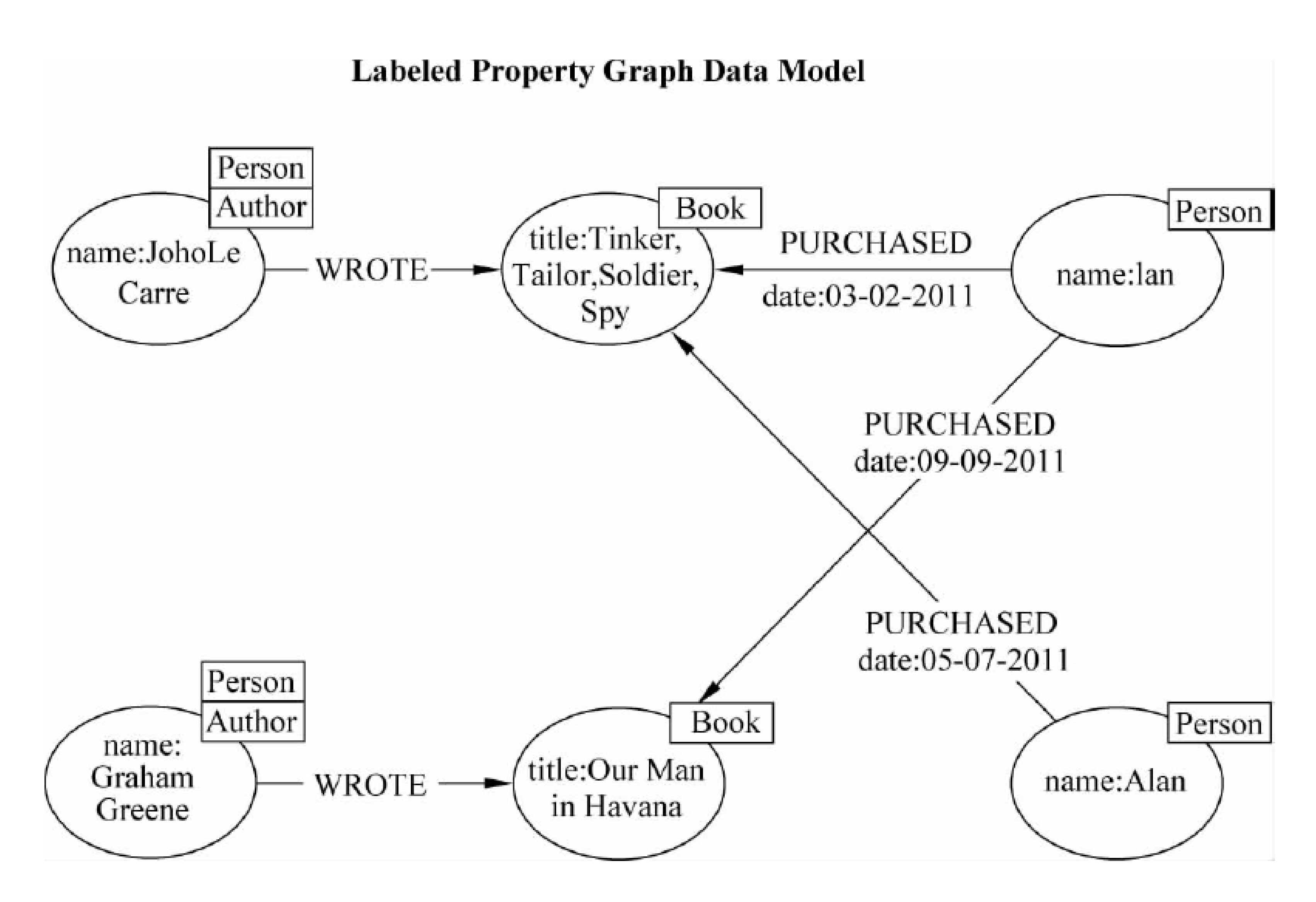


图 4-22　图数据实例

(2) 所有的关系在 Neo4j 中都是同等重要的。

(3) 提供了图的深度优先搜索、广度优先搜索、最短路径、简单路径以及 Dijkstra 等算法。

4.5.5　文档数据库

文档数据库中的文档是一个数据记录，这个记录能够对包含的数据类型和内容进行“自我描述”，如 XML 文档、HTML 文档和 JSON 文档。下面是一个以 JSON 为存储格式的文档数据库实例。

```
{
"ID":1,
"NAME":"SequoiaDB",
"Tel":{
      "Office":"123123","Mobile":"132132132"
   }
"Addr":"China,GZ"
}
```

可以看到，数据是不规则的，每一条记录包含了所有有关“SequoiaDB”的信息而没有任何外部的引用，这条记录就是“自包含”的。这就使得记录很容易完全移动到其他服务

器，因为这条记录的所有信息都包含在里面了，不需要考虑还有信息在其他表没有一起迁移走。同时，因为在移动过程中只有被移动的那一条记录（文档）需要操作而不像关系型中每个有联系的表都需要锁住来保证一致性，这样 ACID 的保证就会变得更快速，读/写的速度也会有很大的提升。

文档数据库中的模型采用的是模型视图控制器（MVC）中的模型层，每个 JSON 文档的 ID 就是它唯一的键，这也大致相当于关系型数据库中的主键。在社交网站领域，文档数据库的灵活性在存储社交网络图片以及内容方面更好，同时并发度也更高。

下面以 MongoDB 这种文档数据库为例讲述文档数据库在实际中的应用。

MongoDB 是一款跨平台、面向文档的数据库。用它创建的数据库可以实现高性能、高可用性，并且能够轻松扩展。MongoDB 的运行方式主要基于两个概念，即集合（collection）与文档（document）。集合就是一组 MongoDB 文档。它相当于关系型数据库（RDBMS）中的表这种概念。集合位于单独的一个数据库中。

（1）集合。集合不能执行模式（schema）。一个集合内的多个文档可以有多个不同的字段。一般来说，集合中的文档都有着相同或相关的目的。

（2）文档。文档就是一组键-值对。文档有着动态的模式，这意味着同一集合内的文档不需要具有同样的字段或结构。

MongoDB 创建数据库采用 use 命令，语法格式为 use DATABASE_NAME，如创建一个 mydb 的数据库：

```
use mydb
```

4.6 阿里云数据库

阿里云数据库是一种稳定可靠、可弹性伸缩的在线数据库服务，提供 NoSQL 数据库和关系型数据库两种数据库服务，并提供可视化数据管理工具和数据库专家服务。下面分别以 Redis、RDS 和 Memcache 几种云数据库具体介绍数据库服务。

4.6.1 云数据库 Redis

阿里云数据库 Redis 版（ApsaraDB for Redis）是兼容开源 Redis 协议的 Key-Value 类型在线存储服务。它支持字符串（String）、链表（List）、集合（Set）、有序集合（SortedSet）、哈希表（Hash）等多种数据类型，以及事务（Transaction）、消息订阅与发布（Pub/Sub）等

高级功能。通过“内存+硬盘”的存储方式,云数据库 Redis 版在提供高速数据读/写能力的同时满足数据持久化需求。

下面是创建和使用 Redis 实例数据的简单步骤。

1. 创建 Redis 实例数据库

相关内容在此省略。

2. 连接 Redis 实例数据库

由于 ApsaraDB for Redis 仅支持从阿里云内网访问,所以此操作方案仅在阿里云 ECS 上执行才生效,因此如果操作 Redis 数据库需要创建一个 ECS 实例,并从 Redis 网站(http://redis.io/)下载一个 Redis 版本到 ECS 服务器上。

连接 ApsaraDB for Redis 必须指定密码,初始密码已在创建实例时创建。

图 4-23 是建立 ECS 实例之后网页控制台终端连接 Redis 实例数据库的方式,该连接的 Redis 实例中缺少密码,所以无法正常操作。

成功连接到实例i-m5e8klh097pl1fik6i69。

```
[qzhong@alibook-ecs redis-3.2.5]$ pwd
/home/qzhong/redis-3.2.5
[qzhong@alibook-ecs redis-3.2.5]$ ./src/redis-cli `cat /tmp/redis-connection.log`
r-bp1585252e97be14.redis.rds.aliyuncs.com:6379> get foo
(error) NOAUTH Authentication required.
r-bp1585252e97be14.redis.rds.aliyuncs.com:6379> set foo bar
(error) NOAUTH Authentication required.
r-bp1585252e97be14.redis.rds.aliyuncs.com:6379> get hello
(error) NOAUTH Authentication required.
r-bp1585252e97be14.redis.rds.aliyuncs.com:6379> get world
(error) NOAUTH Authentication required.
r-bp1585252e97be14.redis.rds.aliyuncs.com:6379> set world world
(error) NOAUTH Authentication required.
r-bp1585252e97be14.redis.rds.aliyuncs.com:6379>
```

图 4-23 ECS 网页控制台连接 Redis 实例

ECS 对应有公网的 IP 地址,也可以利用 bash、xshell 等桌面终端连接到 ECS 服务器上,然后再连接 Redis 实例,如图 4-24 所示。在该图中,tmp 下 redis-connection.log 文件的内容为指定具体的 Redis 主机、Redis 端口以及 Redis 连接密码。

3. 操作 Redis 数据库

云数据库 Redis 支持开源社区中 Redis 的绝大部分功能,但是也有部分受限制,详情可以参考 Redis 命令(https://help.aliyun.com/document_detail/26356.html?spm=5176.7738699.0.0.qx7Pqw)。

图 4-25 是云数据库 Redis 的简单操作,包括 get 操作、set 操作、incr 操作和 del 操作。

```
Connecting to 139.129.219.199:22...
Connection established.
To escape to local shell, press 'Ctrl+Alt+]'.

Last login: Sat Nov 19 11:10:02 2016 from 159.226.43.96

Welcome to aliyun Elastic Compute Service!

[qzhong@alibook-ecs ~]$ cd redis-3.2.5/
[qzhong@alibook-ecs redis-3.2.5]$ pwd
/home/qzhong/redis-3.2.5
[qzhong@alibook-ecs redis-3.2.5]$ ./src/redis-cli `cat /tmp/redis-connection.log`
r-bp1585252e97be14.redis.rds.aliyuncs.com:6379> get foo
"bar"
r-bp1585252e97be14.redis.rds.aliyuncs.com:6379> get world
"world"
r-bp1585252e97be14.redis.rds.aliyuncs.com:6379> get hello
"world"
r-bp1585252e97be14.redis.rds.aliyuncs.com:6379>
```

图 4-24 桌面终端连接 ECS 服务器

```
04e8b91b0ee842ff.redis.rds.aliyuncs.com:6379> set server:name "fido"
OK
04e8b91b0ee842ff.redis.rds.aliyuncs.com:6379> get server:name
"fido"
04e8b91b0ee842ff.redis.rds.aliyuncs.com:6379> set connection 10
OK
04e8b91b0ee842ff.redis.rds.aliyuncs.com:6379> incr connection
(integer) 11
04e8b91b0ee842ff.redis.rds.aliyuncs.com:6379> incr connection
(integer) 12
04e8b91b0ee842ff.redis.rds.aliyuncs.com:6379> del connection
(integer) 1
04e8b91b0ee842ff.redis.rds.aliyuncs.com:6379> incr connection
(integer) 1
04e8b91b0ee842ff.redis.rds.aliyuncs.com:6379> incr connection
(integer) 2
04e8b91b0ee842ff.redis.rds.aliyuncs.com:6379>
```

图 4-25 云数据库 Redis 的操作

4. 性能测试和性能监控

采用 Redis 自带的 redis benchmark 测试 Redis 的性能。执行操作的命令如下：

```
./src/redis-benchmark -h <redis-instance-hosts> -p <redis-port> -a <redis-instance-password>
```

执行上述命令之后的部分结果如图 4-26 所示，性能监控可以通过网页形式查看，图 4-27 所示为系统的吞吐量性能监控。

```
====== MSET (10 keys) ======
  100000 requests completed in 4.79 seconds
  50 parallel clients
  3 bytes payload
  keep alive: 1

0.07% <= 1 milliseconds
25.23% <= 2 milliseconds
88.13% <= 3 milliseconds
96.12% <= 4 milliseconds
98.86% <= 5 milliseconds
99.65% <= 6 milliseconds
99.82% <= 7 milliseconds
99.93% <= 8 milliseconds
99.93% <= 9 milliseconds
99.97% <= 10 milliseconds
99.99% <= 11 milliseconds
100.00% <= 203 milliseconds
100.00% <= 203 milliseconds
20885.55 requests per second
```

图 4-26　性能测试的部分结果

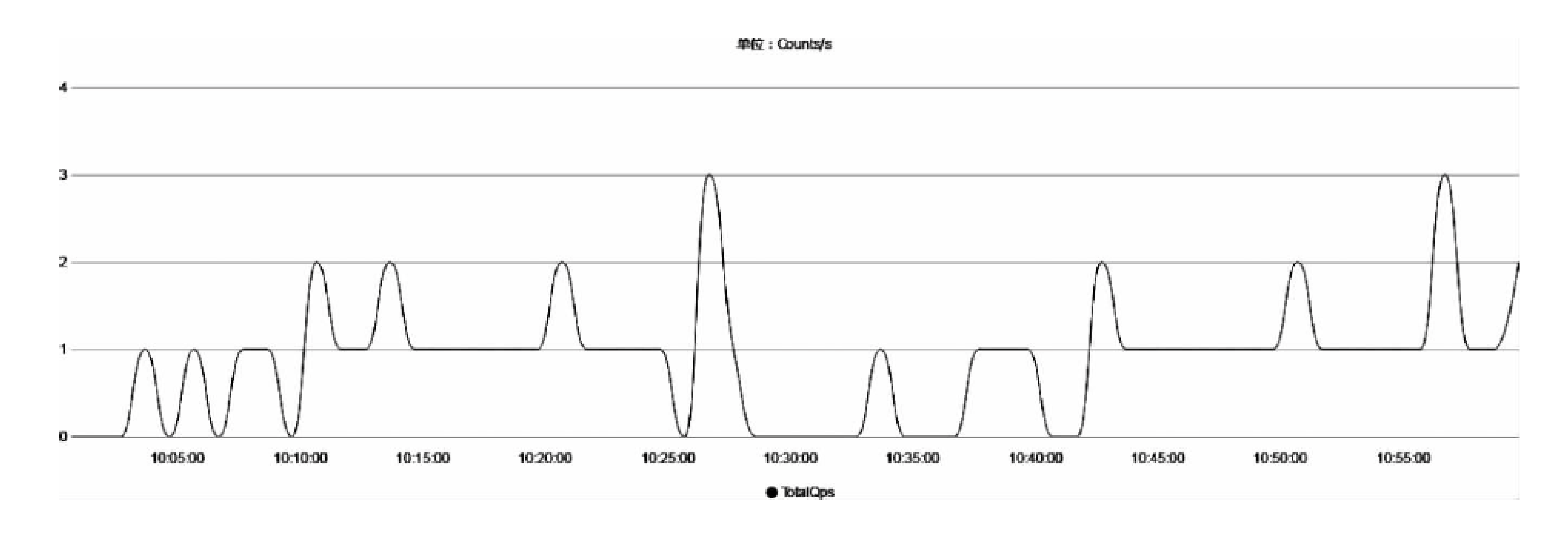


图 4-27　Redis 的吞吐量性能监控

4.6.2　云数据库 RDS

云数据库 RDS(ApsaraDB for RDS,RDS)是一种稳定可靠、可弹性伸缩的在线数据库服务。其基于飞天分布式系统和全 SSD 盘高性能存储,支持 MySQL、SQL Server、PostgreSQL 和 PPAS(高度兼容 Oracle)引擎,默认部署主备架构且提供了容灾、备份、恢复、监控、迁移等方面的全套解决方案。

阿里云数据库 MySQL 版基于阿里巴巴的 MySQL 源码分支,经过双十一高并发、大数据量的考验,拥有优良的性能和吞吐量。除此之外,阿里云数据库 MySQL 版还拥有经过优化的读/写分离、数据压缩、智能调优等高级功能。

下面以阿里云数据库 MySQL 版做简单的使用说明,详细使用文档参考阿里云官方文档(https://help.aliyun.com/document_detail/26124.html?spm=5176.doc26092.6.136.1LL56L)。

1. 设置白名单

为了数据库的安全稳定，用户应该将需要访问数据库的 IP 地址或者 IP 段加入白名单。在启用目标实例前需要先修改白名单。

图 4-28 所示为设置成允许所有网络 IP 访问，设置成 0.0.0.0/0。



图 4-28 设置白名单

2. 账户管理、数据库连接与管理

与正常的 MySQL 操作基本类似。

3. 性能监控

通过网页形式可以监控 MySQL 示例的网络流量、磁盘使用空间、CPU 利用率等，如图 4-29 和图 4-30 所示。

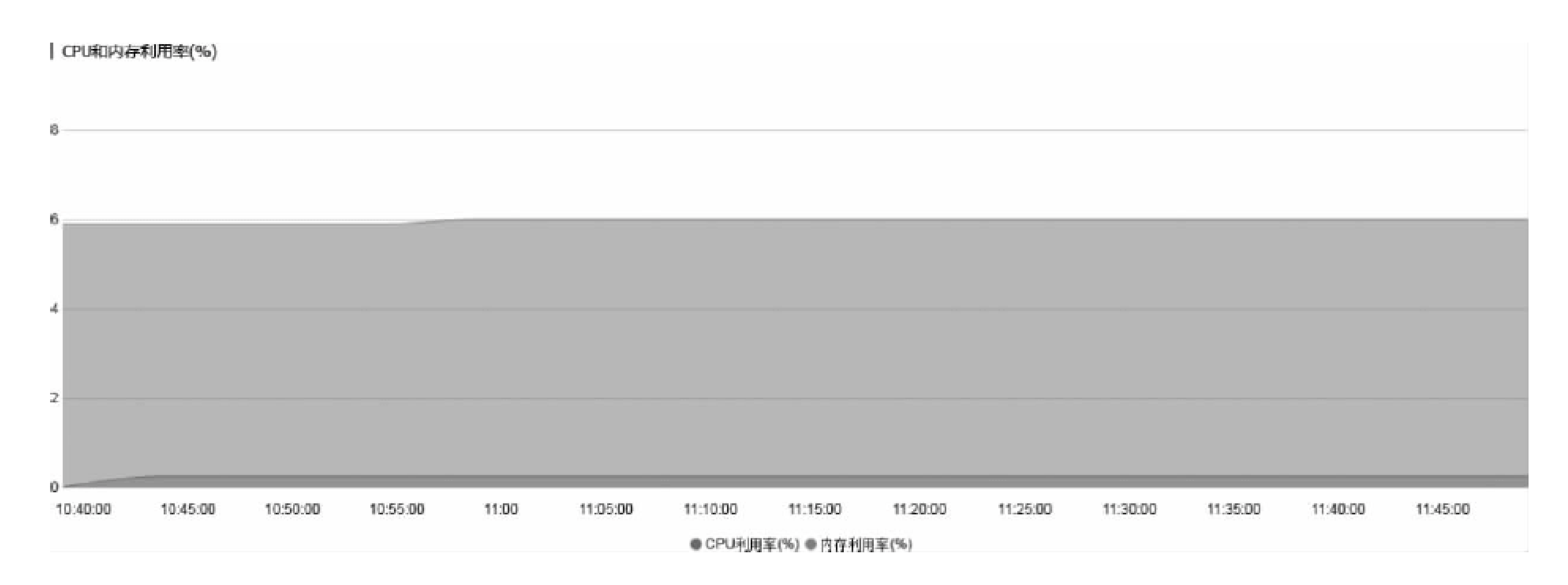


图 4-29 CPU 利用率监控

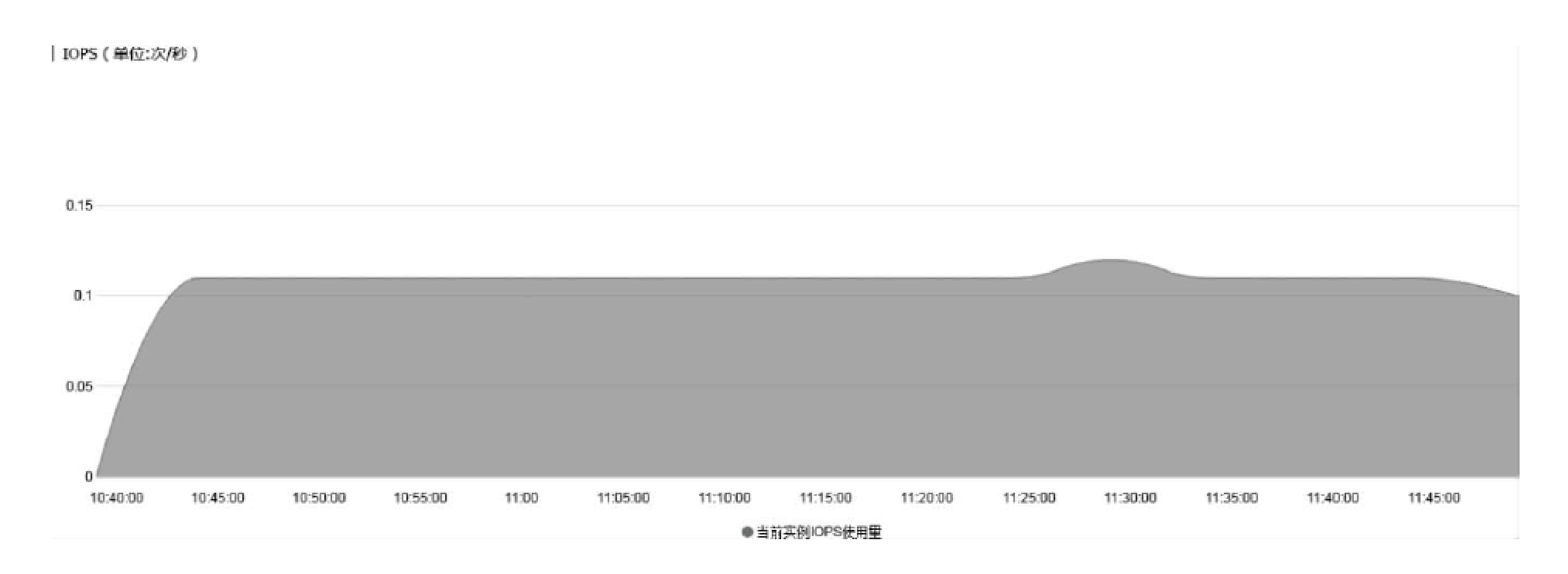


图 4-30　MySQL 系统吞吐量

4.6.3　云数据库 Memcache

云数据库 Memcache 版(ApsaraDB for Memcache)是基于内存的缓存服务，支持海量小数据的高速访问。云数据库 Memcache 可以极大地缓解对后端存储的压力，提高网站或应用的响应速度。云数据库 Memcache 支持 Key-Value 的数据结构，兼容 Memcached 协议的客户端都可与阿里云数据库 Memcache 版进行通信。

云数据库 Memcache 版支持即开即用的方式快速部署；对于动态 Web、APP 应用，可通过缓存服务减轻对数据库的压力，从而提高网站整体的响应速度。任何兼容 Memcached 协议的客户端都可以访问阿里云数据库 Memcache 版服务，用户可以根据自身的应用特点选用支持 SASL 和 Memcached Binary Protocol 的任何 Memcached 客户端。

下面是 Memcache 的简单使用：

1. 环境配置

下载 C++ 客户端(https://launchpad.net/libmemcached/1.0/1.0.18/+download/libmemcached-1.0.18.tar.gz)，下载完成之后编译。

2. 下载 C++ 代码示例

从 GitHub 上下载对应的 aliyunbook 仓库代码(https://github.com/alibook/alibook-bigdata.git)，对应 chapter4 代码部分。

```
cd aliyunbook/chapter4
vim memcached_test.c          //修改 TARGET_HOST 为对应的 Memcached 实例名字
./build.sh                    //编译对应的代码
```

3. 关于云数据库 Memcache 用户名密码访问管理

为了简单，设置云数据库 Memcache 免密码访问，这样对应的 ECS 主机可以无密码访问 Memcache，如图 4-31 所示。

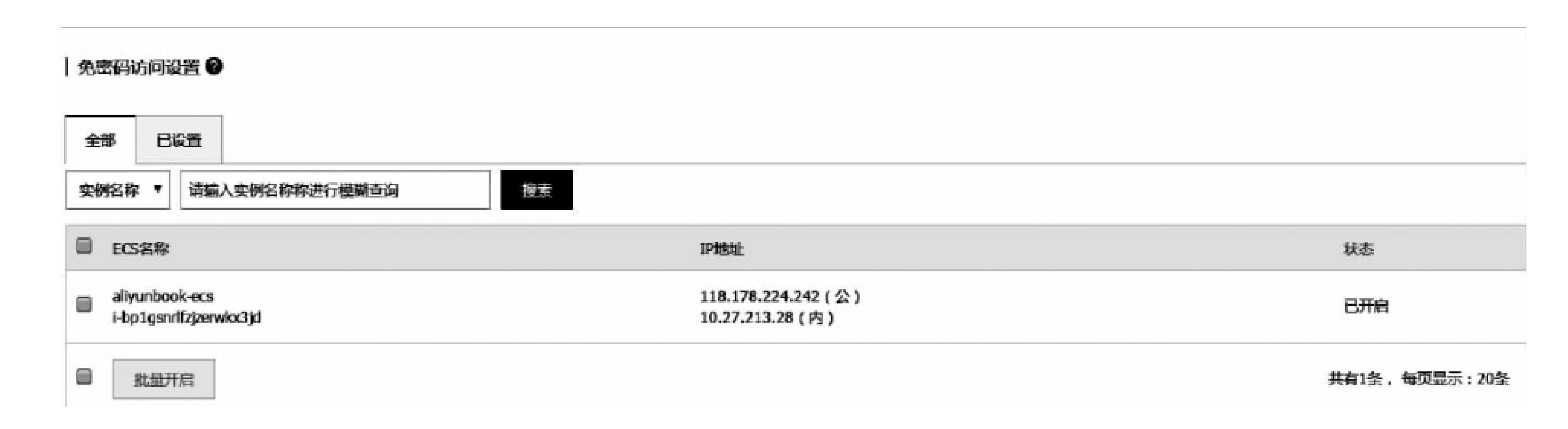


图 4-31 设置无密码访问

4. 运行测试用例

使用“./build.sh”生成 memcached_test.c，对应的 memcached_test.c 代码如下：

```
#include <stdio.h>
#include <string.h>
#include <libmemcached/memcached.h>

#define TARGET_HOST  "5fe328398ee149a2.m.cnqdalicm9pub001.ocs.aliyuncs.com"

int main(int argc, char *argv[])
{
    memcached_st *memc = NULL;
    memcached_return_t rc;
    memcached_server_st *server;
    memc = memcached_create(NULL);
    server = memcached_server_list_append(NULL, TARGET_HOST, 11211,&rc);

    rc = memcached_server_push(memc,server);
    if (rc != MEMCACHED_SUCCESS) {
      fprintf(stderr, "Failed to connect MemCached server, error: %d\n", rc);
    }
    memcached_server_list_free(server);

    const char *key = "TestKey";
```

```
const char * value = "TestValue";
size_t value_length = strlen(value);
size_t key_length = strlen(key);
int expiration = 0;
uint32_t flags = 0;

/* Save data */
fprintf(stdout, "========================================\n");
rc = memcached_set(memc, key, key_length, value, value_length, expiration, flags);
if (rc != MEMCACHED_SUCCESS) {
  fprintf(stderr, "Save data failed: %d\n", rc);
  return -1;
}
fprintf(stdout, "Save data succeed, key: %s value: %s\n" ,key, value);

/* Get data */
fprintf(stdout, "========================================\n");
fprintf(stdout, "Start get key: %s\n", key);
char* result = memcached_get(memc, key, key_length, &value_length, &flags, &rc);
fprintf(stdout, "Get value: %s\n", result);

/* Get data which doesn't exist */
fprintf(stdout, "========================================\n");
const char * key_a = "TestKey2";
size_t      key_a_length = strlen(key_a);
fprintf(stdout, "Start get key: %s\n", key_a);
result = memcached_get(memc, key_a, key_a_length, &value_length, &flags, &rc);
fprintf(stdout, "Get value: %s\n", result);

/* Delete data */
fprintf(stdout, "========================================\n");
fprintf(stdout, "Start delete key: %s\n", key);
rc = memcached_delete(memc,key, key_length, expiration);
if (rc != MEMCACHED_SUCCESS) {
  fprintf(stderr, "Delete key failed: %s\n", rc);
}
fprintf(stdout, "Delete key succeed\n");

/* free */
```

```
    memcached_free(memc);
    return 0;
    }
```

最后运行 ocs_test_sample1，运行结果如图 4-32 所示。

```
[qzhong@alibook-ecs chapter4]$ ls
build.sh  memcached_test  memcached_test.c
[qzhong@alibook-ecs chapter4]$ ./memcached_test
=======================================
Save data succeed, key: TestKey value: TestValue
=======================================
Start get key: TestKey
Get value: TestValue
=======================================
Start get key: TestKey2
Get value: (null)
=======================================
Start delete key: TestKey
Delete key succeed
[qzhong@alibook-ecs chapter4]$
```

图 4-32　Memcached 示例的运行结果

Memcache 监控信息如图 4-33 和图 4-34 所示。

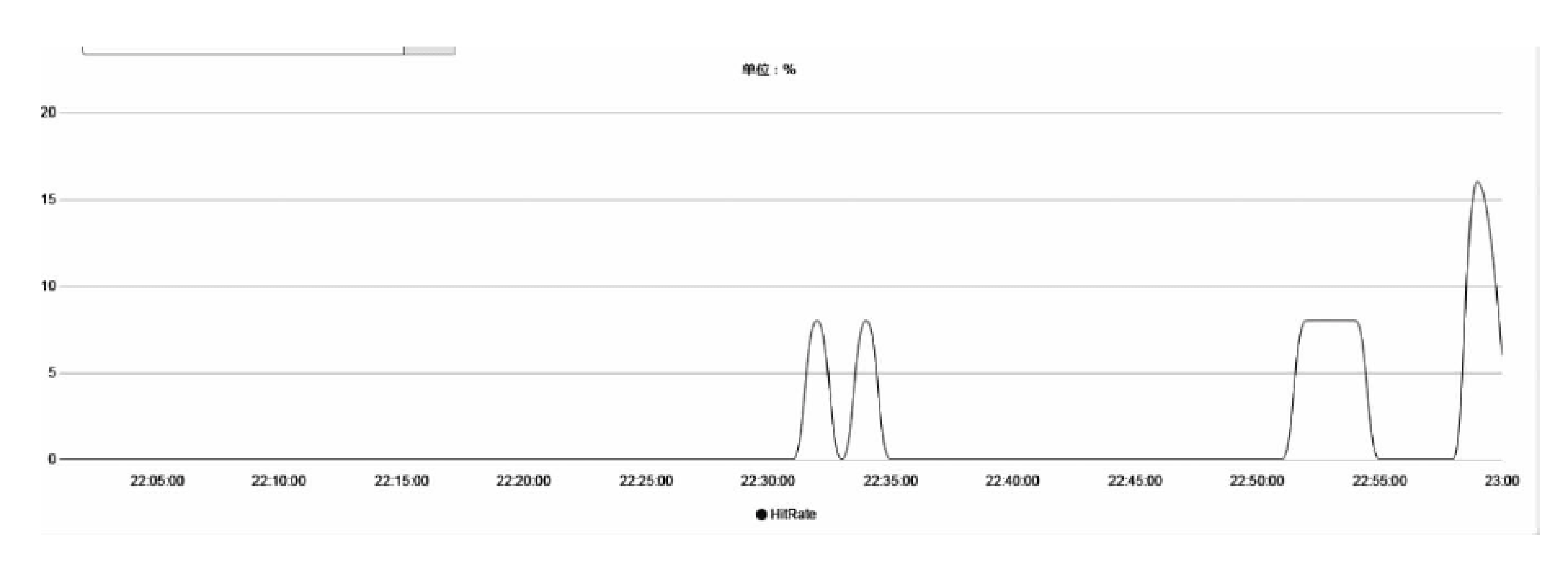


图 4-33　Memcached 命中率监控

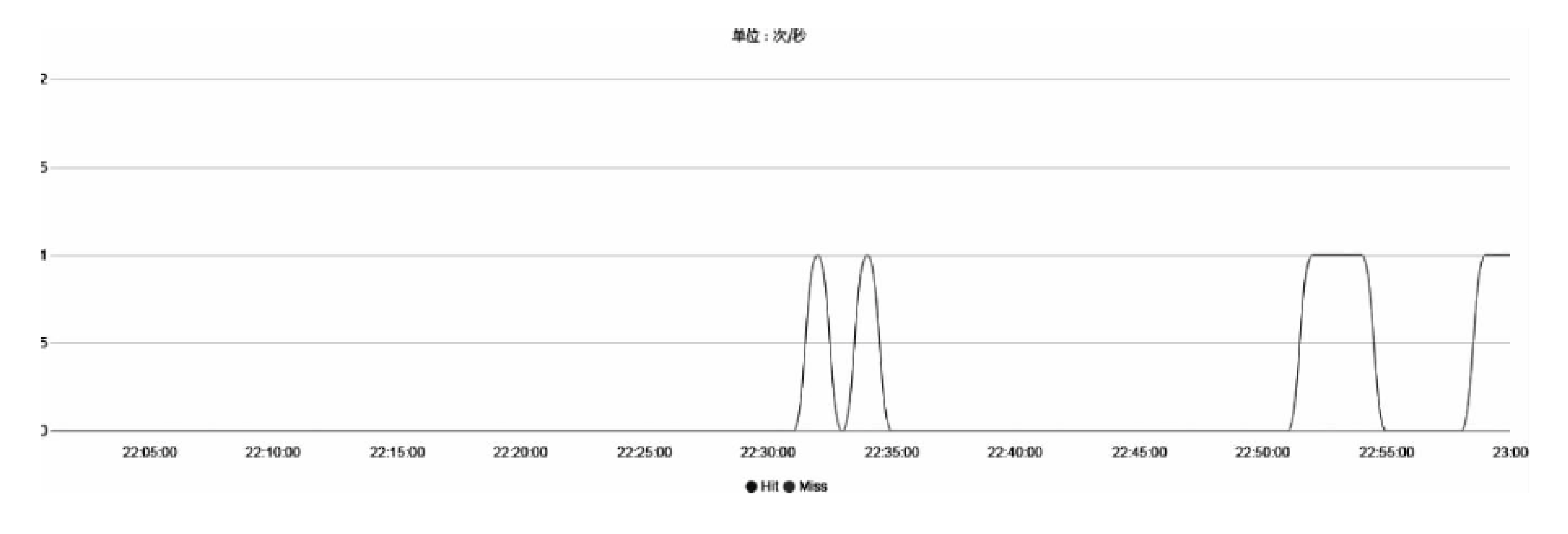


图 4-34　Memcached 请求命中及请求未命中状态

4.7 大数据存储技术的趋势

目前数据在不断产生,需要存储系统提供更强的存储能力以及更高的检索效率。当前硬件设备的成本不断下降,如内存成本在不断降低,为了满足高并发低延迟的需求,不断出现新的内存存储系统,包括 RAMCloud、Mica、VoltDB 等以内存为存储介质的分布式存储系统。因此,以内存为存储介质的分布式存储系统将是以后的一个发展方向。

其次,当前硬件设备的性能不断提升,传统的单纯只考虑软件设计原则显然不适合,需要结合新的硬件特性来做加速和重新设计新的分布式存储系统,比如 NVM 存储介质的出现,在设计以 NVM 为存储介质的分布式系统的时候在可靠性方面必然与易失型内存有显著的区别。同时,多核 CPU、高速网卡等都需要充分发挥性能,因此软/硬件协同也是分布式存储系统的一个技术趋势。

存储系统的通用性和针对性是两个不同的设计选择,目前大多数存储系统都不太可能满足各种场景,比如 HDFS 分布式文件系统是针对大文件存储,Facebook Haystack 是针对小文件,淘宝的 TFS 也是针对于小文件存储的。因此,针对性是分布式存储系统的发展趋势。

4.8 习题

1. 简述数据的分片类型。
2. 简述 LSM Tree 的工作原理。
3. 分布式文件系统存储格式有哪几种?分别阐述。
4. 什么叫 NoSQL?它与关系型数据库有什么区别?简述 NoSQL 的使用场景。
5. 数据一致性包含哪几种?各自有什么区别?

第二部分

大数据处理

第 5 章

分布式处理

5.1 CPU 多核和 POSIX Thread

为了提高任务的计算处理能力，下面分别从硬件和软件层面研究新的计算处理能力。

在硬件设备上，CPU 技术不断发展，出现了 SMP(对称多处理器)和 NUMA(非一致性内存访问)两种高速处理的 CPU 结构。处理器性能的提升给大量的任务处理提供了很大的发展空间。图 5-1 是 SMP 和 NUMA 结构的 CPU，CPU 核数的增加带来了计算能力的提高，但是也随之带来了大量的问题需要解决，比如 CPU 缓存一致性问题、NUMA 内存分配策略等，目前已经有比较不错的解决方案。

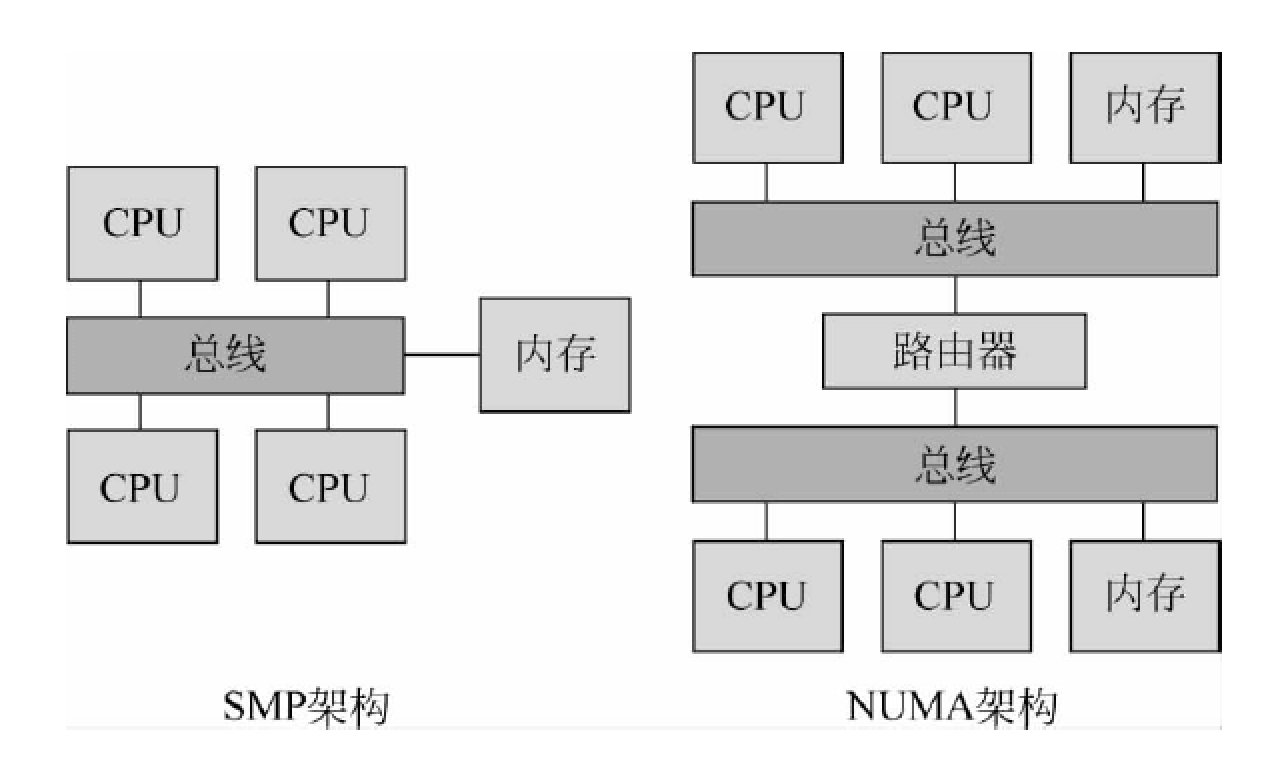


图 5-1　SMP 和 NUMA 架构 CPU

在软件层面出现了多进程和多线程编程。进程是内存资源管理单元,线程是任务调度单元。图 5-2 是进程和线程之间的区别。

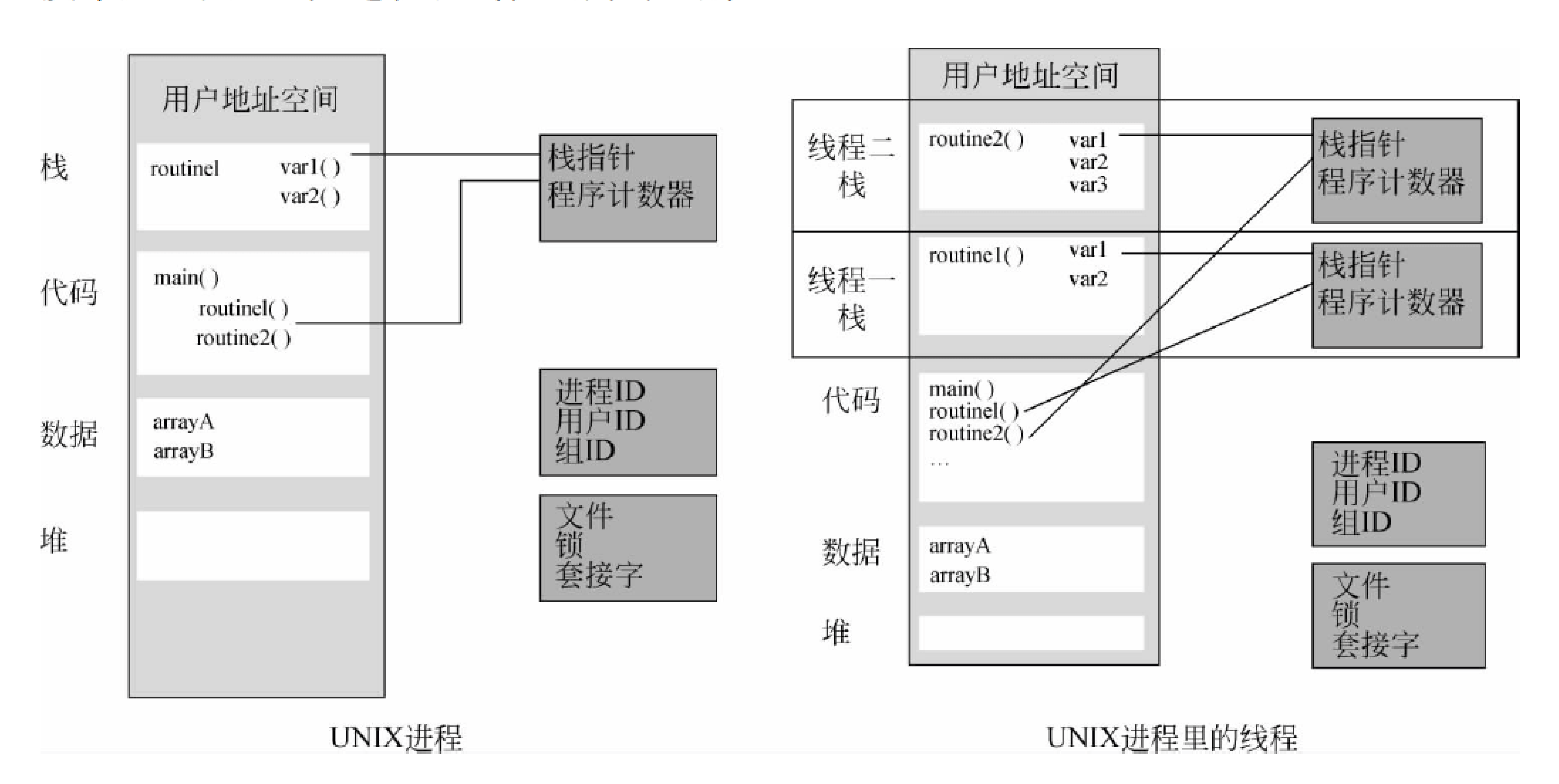


图 5-2 进程与线程

总的来说,线程所占用的资源更少,运行一个线程所需要的资源包括寄存器、栈、程序计数器等。早期不同厂商提供了不同的多线程编写库,这些线程库差异巨大,为了统一多种不同的多线程库,共同制定了 POSIX Thread 多线程编程标准,以充分利用多个不同的线程库。组成 POSIX Thread 的 API 分成以下 4 个大类:

(1) 线程管理。线程管理主要负责线程的 create、detach、join 等,也包括线程属性的查询和设置。

(2) mutexes。处理同步的例程(routine)称为 mutex,mutex 提供了 create、destroy、lock 和 unlock 等函数。

(3) 条件变量。条件变量主要用于多个线程之间的通信和协调。

(4) 同步。同步用于管理读/写锁以及 barriers。

5.2 MPI 并行计算框架

MPI(Message Passing Interface)是一个标准且可移植的消息传递系统,服务于大规模的并行计算。MPI 标准定义了采用 C、C++、Fortran 语言编写程序的函数语法和语义。目前有很多经过良好测试和高效率的关于 MPI 的实现,广泛采用的实现有 MPICH。下面以 MPICH 为例展开对 MPI 的讲解。

MPICH 是一个高性能且可以广泛移植的 MPI 实现。图 5-3 为 MPICH 的架构图。

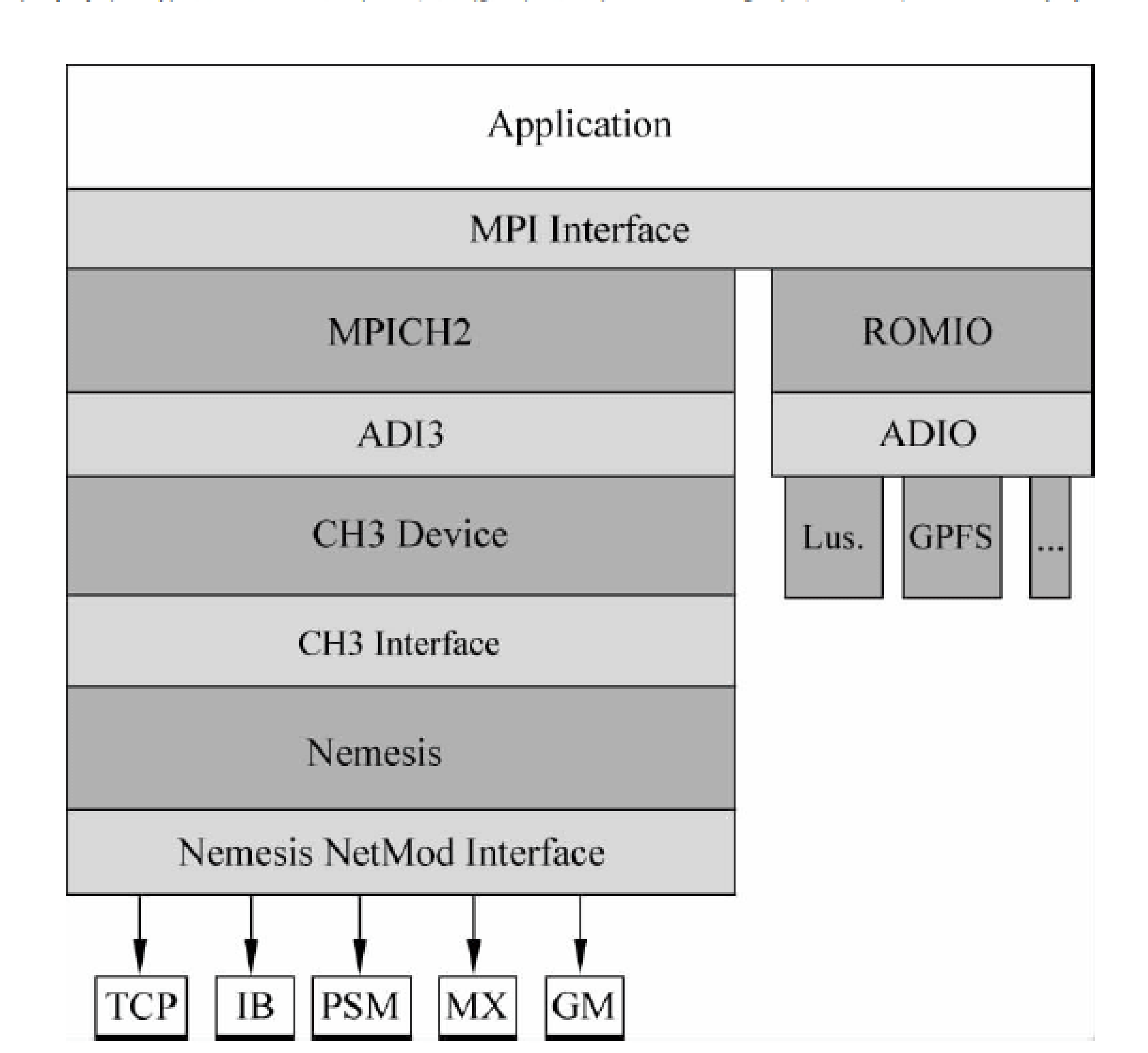


图 5-3　MPICH 架构

如图 5-3 所示，应用程序通过 MPI 结构连接到 MPICH 接口层，图中的 ROMIO 是 MPI-IO 的具体实现版本，对应 MPI 标准中的高性能实现。MPICH 包括 ADI3、CH3 Device、CH3 Interface、Nemesis、Nemesis NetMod Interface。

(1) ADI3。ADI 是抽象设备接口(abstract device interface)，MPICH 通过 ADI3 接口层隔离底层的具体设备。

(2) CH3 Device。CH3 Device 是 ADI3 的一个具体实现，使用了相对少数目的函数功能。在 CH3 Device 实现了多个通信 channel，channel 提供了两个 MPI 进程之间传递数据的途径以及进程通信。当前包括两个 channel，即 Nemesis 和 Sock，其中 Sock 是一个基于 UNIX Socket 的 channel，而 Nemesis 支持多种方法，不仅仅局限于 Socket 通信。

(3) CH3 Interface。CH3 Inferface 用于定义访问 Nemesis 的接口规范。

(4) Nemesis。Nemesis 允许两个 MPI 进程之间的网络通信采取多种方法，包括 TCP、InfiniBand 等。

5.3　Hadoop MapReduce

Hadoop 是一个由 Apache 基金会开发的分布式系统基础架构。Hadoop 框架最核心的设计就是 HDFS 和 MapReduce，HDFS 为海量的数据提供了存储，而 MapReduce 为海量的数据提供了计算。

HDFS(Hadoop Distributed File System)有高容错性的特点,并且设计用来部署在低廉的硬件上;而且它提供高吞吐量来访问应用程序的数据,适合有着超大数据集的应用程序。HDFS放宽了POSIX的要求,可以用流的形式访问文件系统中的数据。

MapReduce是Google公司提出的一个软件框架,用于大规模数据集(大于1TB)的并行运算。"Map"和"Reduce"概念以及它们的主要思想都是从函数式编程语言借来的,还有从矢量编程语言借来的特性。

当前的软件实现是指定一个Map函数,用来把一组键值对映射成一组新的键值对,指定并发的Reduce函数,用来保证所有映射的键值对中的每一个共享相同的键组。

处理流程如下:

(1) MapReduce框架将应用的输入数据切分成 M 个模块,典型的数据块大小为64MB。

(2) 具有全局唯一的主控Master以及若干个Worker,Master负责为Worker分配具体的Map或Reduce任务并做全局管理。

(3) Map任务的Worker读取对应的数据块内容,从数据块中解析Key/Value记录数据并将其传给用户自定义的Map函数,Map函数输出的中间结果Key/Value数据在内存中缓存。

(4) 缓存的Map函数产生的中间结果周期性地写入磁盘,每个Map函数中间结果在写入磁盘前被分割函数切割成 R 份,R 是Reduce的个数。一般用Key对 R 进行哈希取模。Map函数完成对应数据块处理后将 R 个临时文件位置通知Master,Master再转交给Reduce任务的Worker。

(5) Reduce任务Worker接到通知时将Map产生的 M 份数据文件pull到本地(当且仅当所有Map函数完成时Reduce函数才能执行)。Reduce任务根据中间数据的Key对记录进行排序,相同Key的记录聚合在一起。

(6) 所有Map、Reduce任务完成,Master唤醒用户应用程序。

5.4 Spark

Spark是UC Berkeley AMP Lab所开源的类Hadoop MapReduce的通用的并行计算框架,Spark基于Map-Reduce算法实现的分布式计算,拥有Hadoop MapReduce所具有的优点;不同于MapReduce的是中间输出和结果可以保存在内存中,从而不再需要读/写HDFS,因此Spark能更好地适用于数据挖掘与机器学习等需要迭代的Map-Reduce的算法。

Spark 最主要的结构是 RDD(Resilient Distributed Datasets),它表示已被分区、不可变的并能够被并行操作的数据集合,不同的数据集格式对应不同的 RDD 实现。RDD 必须是可序列化的。RDD 可以缓存到内存中,每次对 RDD 数据集操作之后的结果都可以存放到内存中,下一个操作可以直接从内存中输入,省去了 MapReduce 大量的磁盘 I/O 操作。这很适合迭代运算比较常见的机器学习算法、交互式数据挖掘。

与 Hadoop 类似,Spark 支持单节点集群或多节点集群。对于多节点操作,Spark 可以采用自己的资源管理器,也可以采用 Mesos 集群管理器来管理资源。Mesos 为分布式应用程序的资源共享和隔离提供了一个有效平台(参见图 5-4)。该设置允许 Spark 与 Hadoop 共存于节点的一个共享池中。

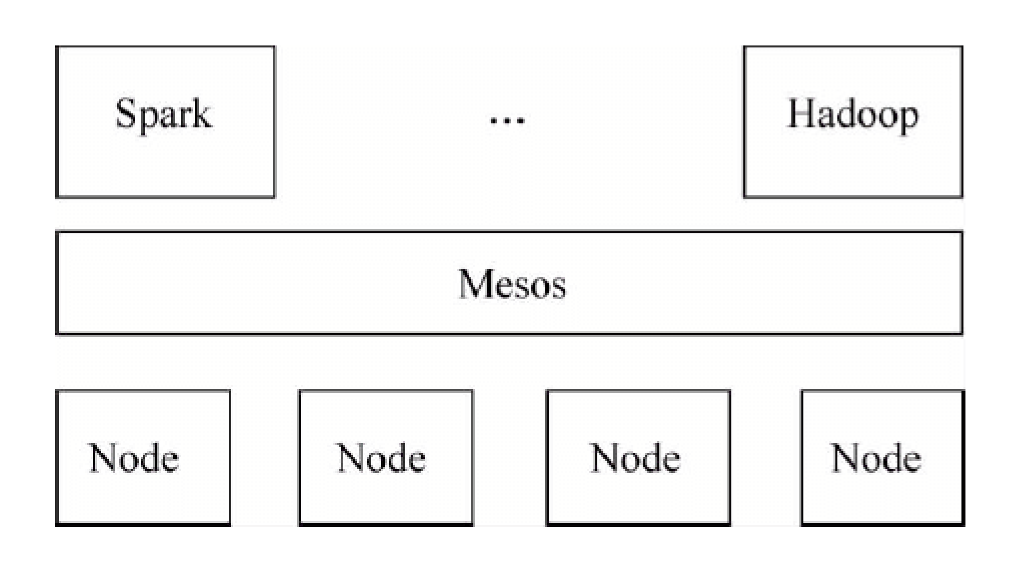


图 5-4 Mesos 集群管理器

5.5 数据处理技术的发展

数据处理从早期的共享分时单 CPU 操作系统处理到多核并发处理,每台计算机设备的处理能力在不断增强,处理的任务复杂度在不断增加,任务的处理时间在不断减少。

然而,随着大数据技术的不断发展,一台计算设备无法胜任目前大数据计算的庞大的计算工作。为了解决单台计算机无法处理大规模数据计算的问题,连接多台计算机设备整合成一个统一的计算系统,对外提供计算服务。早期 Google 公司的分布式计算框架 MapReduce 采用的思想就是连接多台廉价的计算设备,以此来提供进行大规模计算任务的能力。但是 MapReduce 是建立在磁盘之上的并行计算框架,由于机械磁盘本身的局限性,MapReduce 仍然有很大的计算延迟。Spark 提出了把计算结果存放在内存中,利用内存作为存储介质的方法极大地缩短了系统的响应时间,降低了计算任务返回结果的延迟。为了满足大规模机器学习计算任务的需求,也设计了大量的分布式机器学习框架来训练机器模型参数,比如 Parameter Server;针对图计算场合,Google 公司设计实现了 Pregel 图计算框架,用于处理最短路径、Dijstra 等经典图计算任务;为了满足实时计算任务需求,设计实现了流计算框架,比如 Spark Streaming、Storm、Flink 等实时计

算框架。

总之，目前处理技术在往大规模、低延迟方向发展，内存空间的扩大以及内存存储成本的降低给大规模数据处理提供了极好的发展契机。

5.6 习题

1. 简述 CPU 技术的发展趋势。
2. 简述 MPICH 并行计算框架。
3. 简述 MapReduce 的原理。

第6章

Hadoop MapReduce解析

6.1 Hadoop MapReduce 架构

MapReduce 是一种分布式计算框架,能够处理大量数据,并提供容错、可靠等功能,运行部署在大规模计算集群中。

MapReduce 计算框架采用主从架构,由 Client、JobTracker、TaskTracker 组成,如图 6-1 所示。

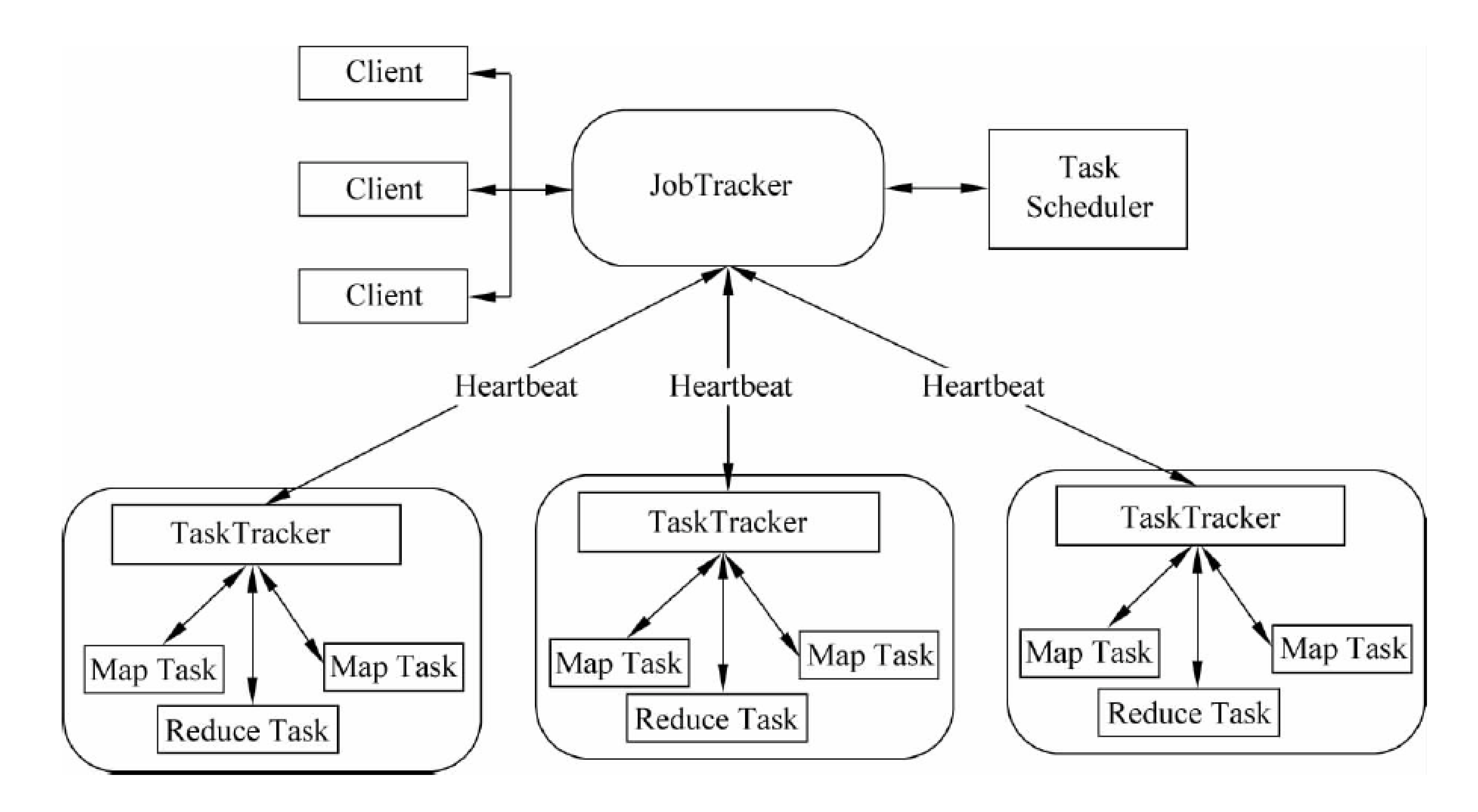


图 6-1　MapReduce 架构

1. Client

用户编写 MapReduce 程序,通过 Client 提交到 JobTracker,由 JobTracker 来执行具体的任务分发。Client 可以在 Job 执行过程中查看具体的任务执行状态以及进度。在 MapReduce 中,每个 Job 对应一个具体的 MapReduce 程序。

2. JobTracker

JobTracker 负责管理运行的 TaskTracker 节点,包括 TaskTracker 节点的加入和退出;负责 Job 的调度与分发,每一个提交的 MapReduce Job 由 JobTracker 安排到多个 TaskTracker 节点上执行;负责资源管理,在当前 MapReduce 框架中每个资源抽象成一个 slot,利用 slot 资源管理执行任务分发。

3. TaskTracker

TaskTracker 节点定期发送心跳信息给 JobTracker 节点,表明该 TaskTracker 节点运行正常。JobTracker 发送具体的任务给 TaskTracker 节点执行。TaskTracker 通过 slot 资源抽象模型,汇报给 JobTracker 节点该 TaskTracker 节点上的资源使用情况,具体分成了 Map slot 和 Reduce slot 两种类型的资源。

在 MapReduce 框架中,所有的程序执行最后都转换成 Task 来执行。Task 分成了 Map Task 和 Reduce Task,这些 Task 都是在 TaskTracker 上启动。图 6-2 显示了 HDFS 作为 MapReduce 任务的数据输入源,每个 HDFS 文件切分成多个 Block,以每个 Block 为单位同时兼顾 Block 的位置信息,将其作为 MapReduce 任务的数据输入源,执行计算任务。

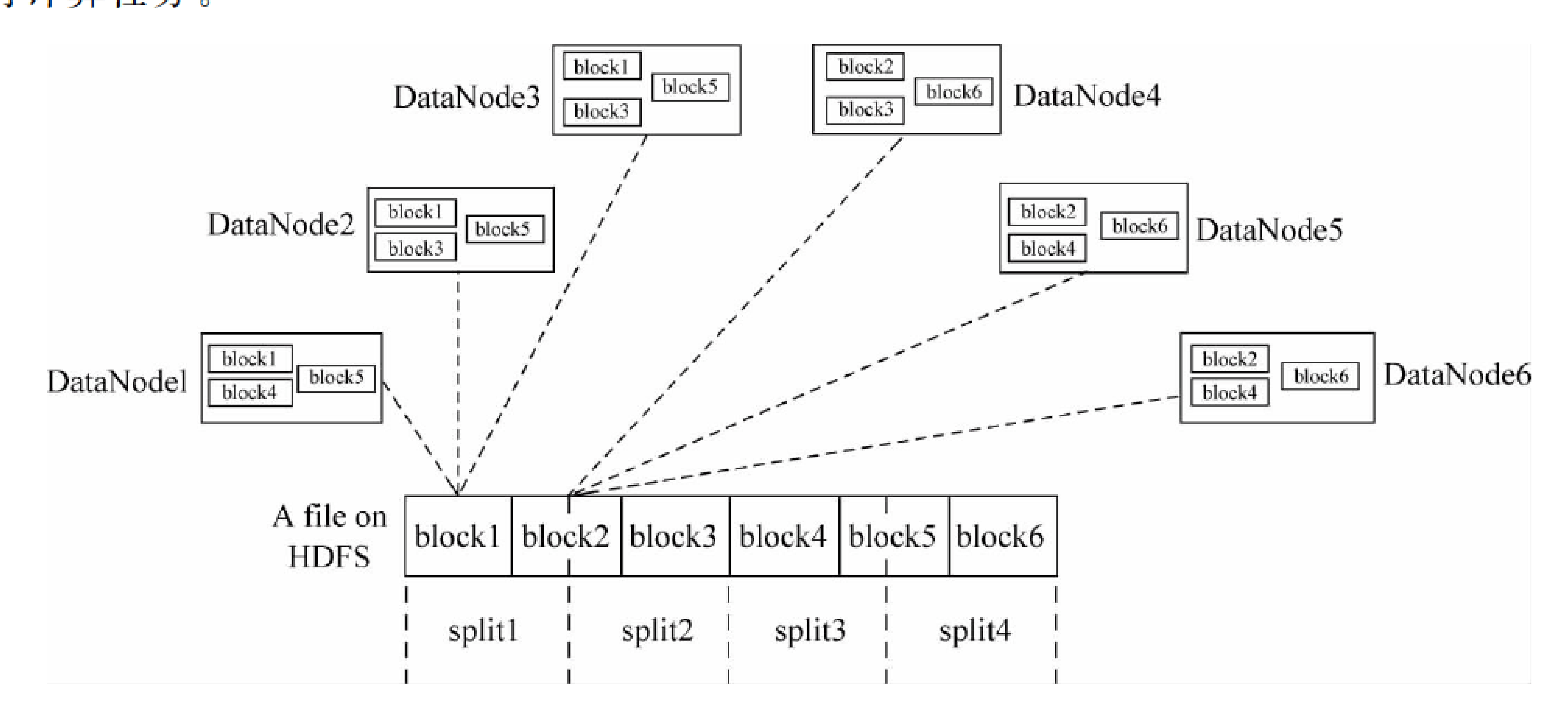


图 6-2 HDFS 作为 MapReduce 任务的数据输入源

6.2 Hadoop MapReduce 与高效能计算、网格计算的区别

在 Hadoop 出现之前，高性能计算和网格计算一直是处理大数据问题主要的使用方法和工具，它们主要采用消息传递接口（Message Passing Interface，MPI）提供的 API 来处理大数据。高性能计算的思想是将计算作业分散到集群机器上，集群计算节点访问存储区域网络 SAN 系统构成的共享文件系统获取数据，这种设计比较适合计算密集型作业。当需要访问像 PB 级别的数据的时候，由于存储设备网络带宽的限制，很多集群计算节点只能空闲等待数据。而 Hadoop 却不存在这种问题，由于 Hadoop 使用专门为分布式计算设计的文件系统 HDFS，在计算的时候只需要将计算代码推送到存储节点上即可在存储节点上完成数据的本地化计算，Hadoop 中的集群存储节点也是计算节点。在分布式编程方面，MPI 属于比较底层的开发库，它赋予了程序员极大的控制能力，但是却要程序员自己控制程序的执行流程、容错功能，甚至底层的套接字通信、数据分析算法等底层细节都需要自己编程实现。这种要求无疑对开发分布式程序的程序员提出了较高的要求。相反，Hadoop 的 MapReduce 却是一个高度抽象的并行编程模型，它将分布式并行编程抽象为两个原语操作，即 Map 操作和 Reduce 操作，开发人员只需要简单地实现相应的接口即可，完全不用考虑底层数据流、容错、程序的并行执行等细节。这种设计无疑大大降低了开发分布式并行程序的难度。

网格计算通常是指通过现有的互联网，利用大量来自不同地域、资源异构的计算机空闲的 CPU 和磁盘来进行分布式存储和计算。这些参与计算的计算机具有分处不同地域、资源异构（基于不同平台，使用不同的硬件体系结构，…）等特征，从而使网格计算和 Hadoop 这种基于集群的计算相区别。Hadoop 集群一般构建在通过高速网络连接的单一数据中心内，集群计算机都具有体系结构、平台一致的特点，而网格计算需要在互联网接入环境下使用，网络带宽等都没有保证。

6.3 MapReduce 工作机制

MapReduce 计算模式的工作原理是把计算任务拆解成 Map 和 Reduce 两个过程来执行，具体如图 6-3 所示。

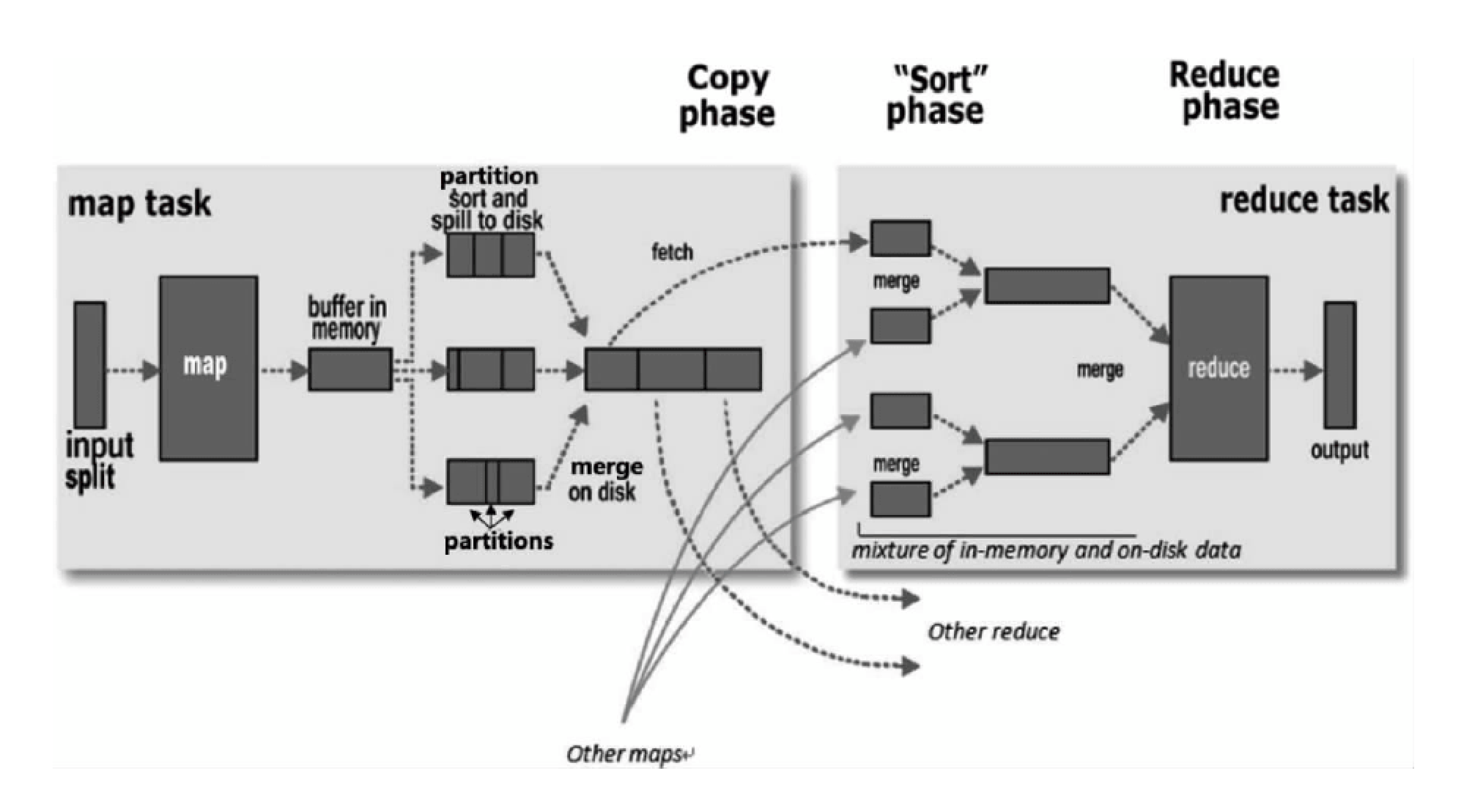


图 6-3 MapReduce 的工作机制

整体而言，一个 MapReduce 程序一般分成 Map 和 Reduce 两个阶段，中间可能会有 Combine。在数据被分割后通过 Map 函数的程序将数据映射成不同的区块，分配给计算机集群处理达到分布式运算的效果，再通过 Reduce 函数的程序将结果汇整，最后输出运行计算结果。

6.3.1 Map

在进行 Map 计算之前，MapReduce 会根据输入文件计算输入分片(input split)，每个输入分片针对一个 Map 任务，输入分片存储的并非数据本身，而是一个分片长度和一个记录数据位置的数组，输入分片往往和 hdfs 的 block(块)的关系很密切。假如设定 hdfs 的块的大小是 64MB，如果我们输入 3 个文件，大小分别是 3MB、65MB 和 127MB，那么 MapReduce 会把 3MB 文件分为一个输入分片，65MB 则是两个输入分片，127MB 也是两个输入分片。换句话说，如果在 Map 计算前做输入分片调整，例如合并小文件，那么会有 5 个 Map 任务将执行，而且每个 Map 执行的数据大小不均，这也是 MapReduce 优化计算的一个关键点。

接着是执行 Map 函数，Map 操作一般由用户指定。Map 函数产生输出结果时并不是直接写入到磁盘，而是采用缓冲方式写入到内存中，并对数据按关键字进行预排序，如图 6-3 所示。每个 Map 任务都有一个环形内存缓冲，用于存储 Map 操作结果，在默认情况下缓冲区大小为 100MB，该值可以用 io. sort. mb 属性修改。当内存中的数据增长到一定比例的时候，可以通过 io. sort. spill. percent 调整参数大小，后台线程会 spill 到磁盘上。在写磁盘的过程中，数据会继续写到内存缓冲区中。

6.3.2　Reduce

执行用户指定的 Reduce 函数，输出计算结果到 HDFS 集群上。Reduce 执行数据的归并，数据是以< key，list（value1，value2…）>的方式存储的。这里以 wordcount 的例子来说明，此时的记录应该是< hadoop，list（1）>、< hello，list（1，1）>、< word，list（1）>，那么结果应该是< hadoop，list（1）>、< hello，list（2）>、< word，list（1）>。

6.3.3　Combine

Combine 是在本地进行的一个在 Map 端做的 Reduce 的过程，其目的是提高 Hadoop 的效率。比如存在两个以 hello 为关键字的记录，直接将数据交给下一个步骤处理，所以在下一个步骤中需要处理两条< hello，1 >的记录，如果先做一次 Combine，则只需处理一次< hello，2 >的记录，这样做的一个好处就是当数据量很大时可以减少很多开销（直接将 partition 后的结果交给 Reduce 处理，由于 TaskTracker 并不一定分布在本节点，过多的冗余记录会影响 I/O，与其在 Reduce 时进行处理，不如在本地先进行一些优化以提高效率）。

6.3.4　Shuffle

Shuffle 描述数据从 Map task 输出到 Reduce task 输入的这段过程。

Map 端的所有工作结束之后，最终生成的这个文件也存放在 TaskTracker 节点的本地文件系统中。每个 Reduce task 不断地通过 RPC 从 JobTracker 那里获取 Map task 是否完成的信息，如果 Reduce task 得到通知，获知某个 TaskTracker 上的 Map task 执行完成。Reduce task 在执行之前的工作就是不断地拉取当前 Job 里每个 Map task 的最终结果，然后对从不同地方拉取过来的数据不断地做 merge，最终形成一个文件作为 Reduce task 的输入文件，如图 6-4 所示。

Reducer 真正运行之前，所有的时间都是在拉取数据，做 merge，且不断重复地做。下面描述 Reduce 端的 Shuffle 细节。

（1）copy 过程。其用于简单地拉取数据。Reduce 进程启动一些数据 copy 线程（Fetcher），通过 HTTP 方式请求 Map task 所在的 TaskTracker 获取 Map task 的输出文件。因为 Map task 早已结束，这些文件就归 TaskTracker 管理在本地磁盘中。

（2）merge 阶段。这里的 merge 如 Map 端的 merge 动作，只是数组中存放的从不同

Map 端 copy 来的数值。copy 来的数据会先放入内存缓冲区中，这里的缓冲区大小要比 Map 端的更为灵活，它基于 JVM 的 heap size 设置，因为在 Shuffle 阶段 Reducer 不运行，所以应该把绝大部分的内存都给 Shuffle 使用。这里需要强调的是，merge 有 3 种形式，即内存到内存、内存到磁盘、磁盘到磁盘。在默认情况下第一种形式不启用，让人比较困惑。当内存中的数据量达到一定阈值时就启动内存到磁盘的 merge。与 Map 端类似，这也是溢写的过程，在这个过程中如果设置有 Combiner，也是会启用的，然后在磁盘中生成了众多的溢写文件。第二种 merge 方式一直在运行，直到没有 Map 端的数据时才结束，然后启动磁盘到磁盘的 merge 方式生成最终的那个文件。

（3）Reducer 的输入文件。不断地 merge，最后会生成一个“最终文件”。那么这里为什么加引号？因为这个文件可能存在于磁盘上，也可能存在于内存中。对用户来说，当然希望将它存放于内存中，直接作为 Reducer 的输入，但默认情况下这个文件是存放于磁盘中的。当 Reducer 的输入文件已定时整个 Shuffle 才最终结束。

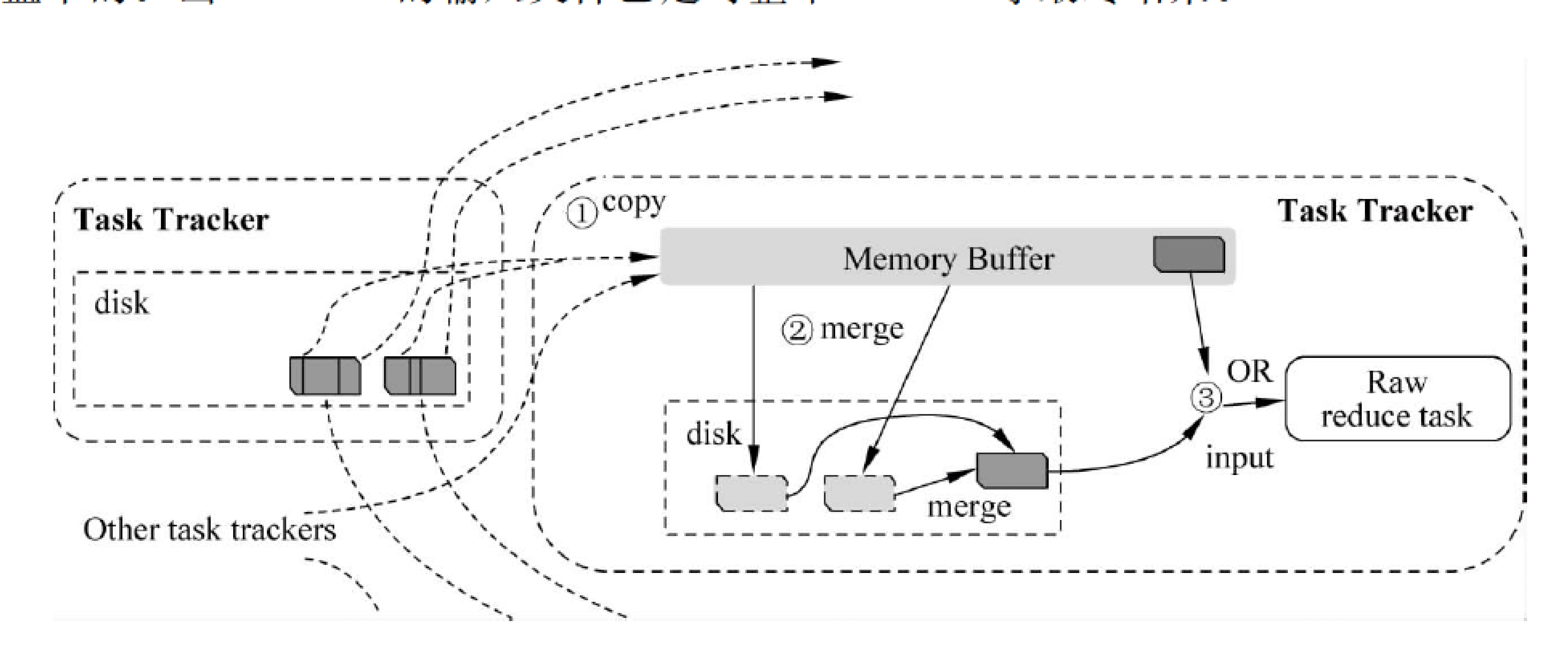


图 6-4 数据从 Map 端 copy 到 Reduce 端

6.3.5 Speculative Task

MapReduce 模型把作业拆分成任务，然后并行运行任务以减少运行时间。存在这样的计算任务，它的运行时间远远长于其他任务的计算任务，减少该任务的运行时间就可以提高整体作业的运行速度，这种任务也称为“拖后腿”任务。导致任务执行缓慢的原因有很多种，包括软件和硬件原因，比如硬件配置更新迭代，MapReduce 任务运行在新旧硬件设备上，负载不均衡，任务调度的局限导致每个计算节点上的任务负载差异较大。

为了解决上述“拖后腿”任务导致的系统性能下降问题，Hadoop 为该 task 启动 Speculative Task，与原始的 task 同时运行，以最快运行结束的结果返回，加快 Job 的执行。当为一个 task 启动多个重复的 task 时，必然导致系统资源的消耗，因此采用

Speculative Task 的方式是一种以空间换时间的方式。

同时启动多个重复的 task 会加速系统资源的竞争，导致 Speculative Task 无法执行。所以启动一个 Speculative Task 需要在一个 Job 的所有 task 都启动完成之后才启动，并且针对那些运行时间比平均运行时间慢的任务。当一个 task 任务完成之后，任何正在运行的重复的任务都会停止。总体来讲，Speculative Task 是优化 MapReduce 计算过程的一个方法。

在 Hadoop 中启动 Speculative Task 的配置方法如下。

```
<property>
    <name>mapred.map.tasks.speculative.execution</name>
    <value>false</value>
</property>
<property>
    <name>mapred.reduce.tasks.speculative.execution</name>
    <value>false</value>
</property>
```

在实际中应该根据具体的情况选择是否需要启动 Speculative Task，因为启动 Speculative Task 是一种加剧资源消耗的过程，会造成系统的性能下降。使用 Speculative Task 的目的是缩短时间，但是以牺牲集群效率为代价。

6.3.6 任务容错

MapReduce 是一种通用的计算框架，有着非常健壮的容错机制，容错粒度包括 JobTracker、TaskTracker、Job、Task、Record 等级别。由于目前 Hadoop 还是单 Master 设计，在一个集群中只有一个 JobTracker，一旦 JobTracker 出现错误往往需要人工介入，但是用户可以通过一些参数进行控制，从而让所有作业恢复运行。TaskTracker 的容错则通过心跳检测、黑名单、灰名单机制对失效的 TaskTracker 节点进行及时处理达到容错效果。同时 Hadoop 还可以通过不同的参数配置来保证 Job、Task 以及 Record 等级别的容错。

用户的一个 MapReduce 作业往往是由很多任务组成的，只有所有的任务执行完毕才算是整个作业成功。对于任务的容错机制，MapReduce 采用最简单的方法进行处理，即“再执行”，也就是说对于失败的任务重新调度执行一次。一般有两种情况需要再执行。

第一种情况：如果是一个 Map 任务或 Reduce 任务失败了，那么调度器会将这个失

败的任务分配到其他节点重新执行。

第二种情况：如果是一个节点死机了，那么在这台死机的节点上已经完成运行的 Map 任务及正在运行中的 Map 和 Reduce 任务都将被调度重新执行，同时在其他机器上正在运行的 Reduce 任务也将被重新执行，这是由于这些 Reduce 任务所需要的 Map 的中间结果数据因为那台失效的机器而丢失了。

6.4 应用案例

下面通过 WordCount、WordMean 等几个例子讲解 MapReduce 的实际应用，编程环境都是以 Hadoop MapReduce 为基础。

6.4.1 WordCount

WordCount 用于计算文件中每个单词出现的次数，非常适合采用 MapReduce 进行处理。处理单词计数问题的思路简单，在 Map 阶段处理每个文本 split 中的数据，产生 < word，1 >这样的键-值对；然后在 Reduce 阶段对相同的关键字求和，最后生成所有的单词计数。重要部分的伪代码如下，详细的可运行代码可以从 GitHub 上下载（https://github.com/alibook/alibook-bigdata.git）。

对应的 Map 端代码如下。

```
public static class TokenizerMapper
        extends Mapper<Object, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context)
        throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
}
```

对应的 Reduce 端代码如下。

```
public static class IntSumReducer
        extends Reducer<Text, IntWritable, Text, IntWritable> {
    public void reduce(Text key, Iterable<IntWritable> values,
        Context context) throws IOException, InterruptedException{
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }

        context.write(key, new IntWritable(sum));
    }

}
```

在主函数中设置该 WordCount Job 的相关环境，包括输入和输出、Map 类和 Reduce 类，如下所示。

```
Configuration conf = new Configuration();
Job job = Job.getInstance(conf, "word count");

job.setJarByClass(WordCount.class);
job.setMapperClass(TokenizerMapper.class);
job.setReducerClass(IntSumReducer.class);
job.setCombinerClass(IntSumReducer.class);

job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);

FileInputFormat.addInputPath(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));

System.exit(job.waitForCompletion(true) ? 0 : 1);
```

WordCount 运行示意图如图 6-5 所示。

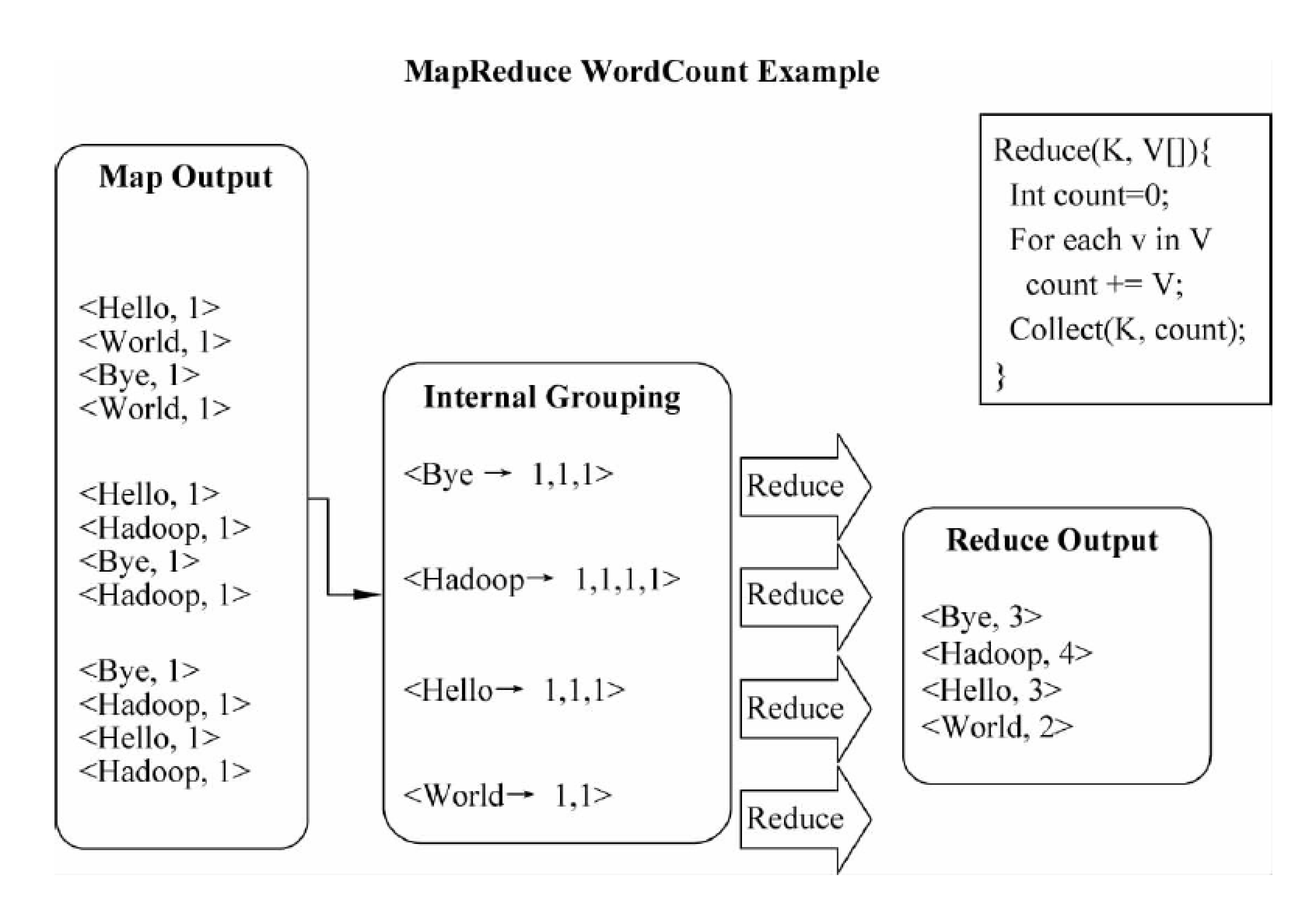


图 6-5　WordCount 运行过程

在终端环境中运行以下命令。

```
bin/hadoop jar /home/qzhong/hadoop - 0. 0. 1. jar alibook. hadoop. WordCount /user/qzhong/
input /user/qzhong/output
```

输入数据存放在 HDFS 集群中，如图 6-6 所示。

```
-rw-r--r--   3 liujun supergroup       4436 2016-11-20 00:28 /user/qzhong/input/capacity-scheduler.xml
-rw-r--r--   3 liujun supergroup       1335 2016-11-20 00:28 /user/qzhong/input/configuration.xsl
-rw-r--r--   3 liujun supergroup        318 2016-11-20 00:28 /user/qzhong/input/container-executor.cfg
-rw-r--r--   3 liujun supergroup       1358 2016-11-20 00:28 /user/qzhong/input/core-site.xml
-rw-r--r--   3 liujun supergroup        591 2016-11-20 00:28 /user/qzhong/input/deliver.exp
-rw-r--r--   3 liujun supergroup       1163 2016-11-20 00:28 /user/qzhong/input/deliver.sh
-rw-r--r--   3 liujun supergroup        109 2016-11-20 00:28 /user/qzhong/input/deliver_list
-rw-r--r--   3 liujun supergroup       3670 2016-11-20 00:28 /user/qzhong/input/hadoop-env.cmd
-rw-r--r--   3 liujun supergroup       4340 2016-11-20 00:28 /user/qzhong/input/hadoop-env.sh
-rw-r--r--   3 liujun supergroup       2490 2016-11-20 00:28 /user/qzhong/input/hadoop-metrics.properties
-rw-r--r--   3 liujun supergroup       2598 2016-11-20 00:28 /user/qzhong/input/hadoop-metrics2.properties
-rw-r--r--   3 liujun supergroup       9683 2016-11-20 00:28 /user/qzhong/input/hadoop-policy.xml
-rw-r--r--   3 liujun supergroup       1580 2016-11-20 00:28 /user/qzhong/input/hdfs-site.xml
-rw-r--r--   3 liujun supergroup       1449 2016-11-20 00:28 /user/qzhong/input/httpfs-env.sh
-rw-r--r--   3 liujun supergroup       1657 2016-11-20 00:28 /user/qzhong/input/httpfs-log4j.properties
-rw-r--r--   3 liujun supergroup         21 2016-11-20 00:28 /user/qzhong/input/httpfs-signature.secret
-rw-r--r--   3 liujun supergroup        620 2016-11-20 00:28 /user/qzhong/input/httpfs-site.xml
-rw-r--r--   3 liujun supergroup       3523 2016-11-20 00:28 /user/qzhong/input/kms-acls.xml
-rw-r--r--   3 liujun supergroup       1325 2016-11-20 00:28 /user/qzhong/input/kms-env.sh
-rw-r--r--   3 liujun supergroup       1631 2016-11-20 00:28 /user/qzhong/input/kms-log4j.properties
-rw-r--r--   3 liujun supergroup       5511 2016-11-20 00:28 /user/qzhong/input/kms-site.xml
-rw-r--r--   3 liujun supergroup      11291 2016-11-20 00:28 /user/qzhong/input/log4j.properties
-rw-r--r--   3 liujun supergroup        938 2016-11-20 00:28 /user/qzhong/input/mapred-env.cmd
-rw-r--r--   3 liujun supergroup       1383 2016-11-20 00:28 /user/qzhong/input/mapred-env.sh
-rw-r--r--   3 liujun supergroup       4113 2016-11-20 00:28 /user/qzhong/input/mapred-queues.xml.template
```

图 6-6　WordCount 输入数据

图 6-7 所示为 WordCount 运行结果，运行结果产生了一个 part-r-00000 文件，保存运算结果。

```
[liujun@dell122 hadoop-2.6.4]$ bin/hadoop jar /home/qzhong/hadoop-0.0.1.jar alibook.hadoop.WordCount /user/qzhong/input /user/qzhong/output
16/11/20 23:14:02 INFO client.RMProxy: Connecting to ResourceManager at dell122/10.61.2.122:8032
16/11/20 23:14:02 WARN mapreduce.JobResourceUploader: Hadoop command-line option parsing not performed. Implement the Tool interface and execute your applica
tion with ToolRunner to remedy this.
16/11/20 23:14:03 INFO input.FileInputFormat: Total input paths to process : 33
16/11/20 23:14:03 INFO mapreduce.JobSubmitter: number of splits:33
16/11/20 23:14:03 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1477880581089_0019
16/11/20 23:14:03 INFO impl.YarnClientImpl: Submitted application application_1477880581089_0019
16/11/20 23:14:03 INFO mapreduce.Job: The url to track the job: http://dell122:8088/proxy/application_1477880581089_0019/
16/11/20 23:14:03 INFO mapreduce.Job: Running job: job_1477880581089_0019
16/11/20 23:14:07 INFO mapreduce.Job: Job job_1477880581089_0019 running in uber mode : false
16/11/20 23:14:07 INFO mapreduce.Job:  map 0% reduce 0%
16/11/20 23:14:11 INFO mapreduce.Job:  map 15% reduce 0%
16/11/20 23:14:12 INFO mapreduce.Job:  map 91% reduce 0%
16/11/20 23:14:14 INFO mapreduce.Job:  map 100% reduce 0%
16/11/20 23:14:16 INFO mapreduce.Job:  map 100% reduce 100%
16/11/20 23:14:16 INFO mapreduce.Job: Job job_1477880581089_0019 completed successfully
16/11/20 23:14:16 INFO mapreduce.Job: Counters: 51
```

图 6-7　WordCount 运行结果

6.4.2　WordMean

下面对 WordCount 稍作修改，改成计算所有文件中单词的平均长度，单词长度的定义是单词的字符个数。现在 HDFS 集群中有大量的文件，需要统计所有文件中所出现单词的平均长度。

其处理也可以采用 MapReduce 方式，计算结果最后以 HDFS 文件的方式保存，保存内容格式为两行数据：第一行是< count，个数>键-值对，为统计出现的所有单词个数；第二行是< length，总长度>键-值对，为统计文件中所有的单词长度。然后从 HDFS 文件中读取 MapReduce 计算结果，求取单词长度的平均值。在 MapReduce 计算过程中，Map 阶段读取每个文件的 split 数据，生成< count，1 >和< length，单词长度>键-值对；Reduce 阶段对相同的 count 关键字和 length 关键字对进行求和。下面是 Map 过程和 Reduce 过程的代码，详细的代码可以从 GitHub 上下载(https://github.com/alibook/alibook-bigdata.git)。

Map 端对应的代码如下。

```
/**
  * Maps words from line of text into 2 key - value pairs;
  * one key - value pair for
  * counting the word, another for counting its length.
  */
public static class WordMeanMapper extends Mapper<Object, Text, Text, LongWritable> {
      private LongWritable wordlen = new LongWritable();

    /**
      * Emits 2 key - value pairs for counting the word and its
      * length. Outputs are(Text, LongWritable).
      *
```

```
         * @param value
         * This will be a line of text coming in from our input file.
         */
        public void map(Object key, Text value, Context context)
               throws IOException, InterruptedException {
         StringTokenizer iter = new StringTokenizer(value.toString());
         while (iter.hasMoreTokens()) {
               wordlen.set(iter.nextToken().length());
               context.write(LENGTH, wordlen);
               context.write(COUNT, ONE);
         }
       }
   }
```

Reduce 端对应的代码如下。

```
   /**
    * Performs integer summation of all the values for each key.
    */
        public static class WordMeanReducer extends Reducer < Text, LongWritable, Text,
   LongWritable> {
       private LongWritable sum = new LongWritable(0);

      /**
       * Sums all the individual values within the iterator and writes
       * them to the same key.
       *
       * @param key
       * This will be one of 2 constants: LENGTH_STR or COUNT_STR.
       * @param values
       * This will be an iterator of all the values associated with that
       * key.
       */
        public void reduce(Text key, Iterable<LongWritable> values, Context context)
            throws IOException, InterruptedException {
              int theSum = 0;
              for (LongWritable value : values) {
                  theSum += value.get();
```

```
            }
            sum.set(theSum);
            context.write(key, sum);
        }
    }
```

在终端运行以下命令。

```
bin/hadoop jar /home/qzhong/wordmean - 0. 0. 1. jar \ alibook. wordmean. WordMean /user/
qzhong/ input  \ /user/qzhong/wordmeanoutput
```

上述命令表示在文件中计算单词的平均长度，计算结果输出到/user/qzhong/wordmeanoutput 中。在该实验中采用和 WordCount 同样的实验数据，运行结果如下所示。

```
The mean length is: 8.360264105642257
```

图 6-8 是 WordMean 运行 MapReduce 产生的计算结果。

```
[liujun@dell122 hadoop-2.6.4]$ bin/hdfs dfs -cat /user/qzhong/wordmeanoutput/part-r-00000
count   8330
length  69641
[liujun@dell122 hadoop-2.6.4]$ bin/hdfs dfs -ls /user/qzhong/wordmeanoutput/part-r-00000
-rw-r--r--   3 liujun supergroup         24 2016-11-20 23:49 /user/qzhong/wordmeanoutput/part-r-00000
[liujun@dell122 hadoop-2.6.4]$
```

图 6-8 WordMean 计算结果

6.4.3 Grep

还是进行大规模文本中单词的相关操作，现在希望提供类似 Linux 系统中 Grep 命令的功能，找出匹配目标串的所有文件，并统计出每个文件中出现目标字符串的个数。

仍然采用 MapReduce 的计算方法提取出匹配目标字符串的所有文件。思路很简单，在 Map 阶段根据提供的文件 split 信息、给定的每个字符串输出< filename，1 >这样的键-值对信息；然后在 Reduce 阶段根据 filename 对 Map 阶段产生的结果进行合并，最后得出匹配目标串的所有文件 grep 信息。

下面是对应的 Map 端和 Reduce 端代码，详细的可运行代码从 GitHub 上下载（https://github.com/alibook/alibook-bigdata.git）。

Map 端代码如下。

```
public static class GrepMapper extends Mapper<Object, Text, Text, IntWritable> {

    public void map(Object obj, Text text, Context context)
                throws IOException, InterruptedException {
    String pattern = context.getConfiguration().get("grep");

    String str = text.toString();
    Pattern r = Pattern.compile(pattern);
    Matcher matcher = r.matcher(str);

    while (matcher.find()) {
      FileSplit split = (FileSplit)context.getInputSplit();
      String filename = split.getPath().getName();

      context.write(new Text(filename), new IntWritable(1));
    }
  }
}
```

Reduce 端代码如下。

```
public static class GrepReducer extends Reducer<Text, IntWritable, Text, IntWritable> {

  public void reduce(Text text, Iterable<IntWritable> values,  Context context)
      throws IOException, InterruptedException{
   int sum = 0;
   Iterator<IntWritable> iterator = values.iterator();
   while (iterator.hasNext()) {
      sum += iterator.next().get();
   }

   context.write(text, new IntWritable(sum));
   }
}
```

在终端运行以下命令。

```
bin/hadoop jar /home/qzhong/grep-0.0.1.jar alibook.grep.Grep hadoop /user/qzhong/input
/user/qzhong/grepoutput
```

上述命令是在所有输入文件中找出匹配 Hadoop 字符串的所有文件,并将计算结果输出到/user/qzhong/grepoutput 目录中。

该命令的运行结果如图 6-9 所示。

```
capacity-scheduler.xml  1
core-site.xml   6
hadoop-env.cmd  5
hadoop-env.sh   7
hadoop-metrics.properties       20
hadoop-metrics2.properties      6
hdfs-site.xml   4
httpfs-log4j.properties 2
httpfs-signature.secret 1
kms-acls.xml    9
kms-log4j.properties    2
kms-site.xml    19
log4j.properties        82
mapred-env.sh   1
yarn-env.cmd    5
yarn-env.sh     4
yarn-site.xml   1
```

图 6-9　Grep 的运行结果

6.5　MapReduce 的缺陷与不足

MapReduce 是一种离线处理框架,比较适合大规模的离线数据处理。在实际的工作环境中,MapReduce 这套分布式处理框架常用于分布式 Grep、分布式排序、Web 访问日志分析、反向索引构建、文档聚类、机器学习、数据分析、基于统计的机器翻译和生成整个搜索引擎的索引等大规模数据处理工作。但是 MapReduce 在实时处理性能方面比较薄弱,不适合处理事务或者单一处理请求。

6.6　习题

1. 简述 MapReduce 架构。
2. 简述 MapReduce 与网格计算、高效能计算之间的区别。
3. 简述 MapReduce 中的 Shuffle 过程。
4. 在 MapReduce 中为什么需要建立 Speculative Task? 会带来哪些问题?
5. 简述 MapReduce 的不足。

第 7 章 Spark解析

7.1 Spark RDD

Spark 是一个高性能的内存分布式计算框架，具备可扩展性、任务容错等特性。每个 Spark 应用都是由一个 driver program 构成，该程序运行用户的 main 函数，同时在一个集群中的节点上运行多个并行操作。Spark 提供的一个主要抽象就是 RDD（Resilient Distributed Datasets），这是一个分布在集群中多节点上的数据集合，利用内存和磁盘作为存储介质，其中内存为主要数据存储对象，支持对该数据集合的并发操作。用户可以使用 HDFS 中的一个文件来创建一个 RDD，可以控制 RDD 存放于内存中还是存储于磁盘等永久性存储介质中。

RDD 的设计目标是针对迭代式机器学习。由于迭代式机器学习本身的特点，每个 RDD 是只读的、不可更改的。根据记录的操作信息，丢失的 RDD 数据信息可以从上游的 RDD 或者其他数据集 Datasets 创建，因此 RDD 提供容错功能。

有两种方式创建一个 RDD：在 driver program 中并行化一个当前的数据集合；或者利用一个外部存储系统中的数据集合创建，比如共享文件系统 HDFS，或者 HBase，或者其他任何提供了 Hadoop InputFormat 格式的外部数据存储。

1. 并行化数据集合（Parallelized Collection）

并行化数据集合可以在 driver program 中调用 JavaSparkContext's parallelize 方法创建，复制集合中的元素到集群中形成一个分布式的数据集 Distributed Datasets。以下

是一个创建并行化数据集合的例子，包含数字 1～5：

```
List<Integer> data = Arrays.asList(1, 2, 3, 4, 5);
JavaRDD<Integer> distData = sc.parallelize(data);
```

一旦上述的 RDD 创建，分布式数据集 RDD 就可以并行操作了。例如可以调用 distData.reduce((a, b) -> a + b)对列表中的所有元素求和。

2. 外部数据集(External Datasets)

Spark 可以从任何 Hadoop 支持的外部数据源创建 RDD，包括本地文件系统、HDFS、Cassandra、HBase、Amazon S3 等。以下是从一个文本文件中创建 RDD 的例子：

```
JavaRDD<String> distFile = sc.textFile("data.txt");
```

一旦创建，distFile 就可以执行所有的数据集操作。

RDD 支持多种操作，分为下面两种类型：

(1) transformation。其用于从以前的数据集中创建一个新的数据集。

(2) action。其返回一个计算结果给 driver program。

在 Spark 中所有的 transformation 都是懒惰的(lazy)，因为 Spark 并不会立即计算结果，Spark 仅仅记录所有对 file 文件的 transformation。以下是一个简单的 transformation 的例子：

```
JavaRDD<String> lines = sc.textFile("data.txt");
JavaRDD<Integer> lineLengths = lines.map(s -> s.length());
int totalLength = lineLengths.reduce((a, b) -> a + b);
```

利用文本文件 data.txt 创建一个 RDD，然后利用 lines 执行 Map 操作，这里 lines 其实是一个指针，Map 操作计算每个 string 的长度，最后执行 reduce action，这时返回整个文件的长度给 driver program。

7.2 Spark 与 MapReduce 的对比

Spark 作为新一代的大数据计算框架，针对的是迭代式计算、实时数据处理，要求处理的时间更少。与 MapReduce 对比整体反映如下：

（1）在中间计算结果方面。Spark 要求计算结果快速返回，处理任务低延迟，因此 Spark 基本把数据存放在内存中，只有在内存资源不够的时候才写到磁盘等存储介质中，同时用户可以指定数据是否缓存在内存中；而 MapReduce 计算过程中 Map 任务产生的计算结果存放到本地磁盘中，由后面需要计算的 Reduce 任务 fetch。

（2）在计算模型方面。Spark 采用 DAG 图描述计算任务，所有的 RDD 操作最后都采用 DAG 描述，然后优化分发到各个计算节点上运行，因此 Spark 拥有更丰富的功能；MapReduce 则只采用 Map 和 Reduce 两个函数，计算功能比较简单。

（3）在计算速度方面。Spark 采用内存作为计算结果主要存储介质，而 MapReduce 采用本地磁盘作为中间结果存储介质，因此 Spark 的计算速度更快。

（4）在容错方面。Spark 采用了和 MapReduce 类似的方式，针对丢失和无法引用的 RDD，Spark 采用利用记录的 transformation，采取重新做已做过的 transformation。

（5）在计算成本方面。Spark 是把 RDD 主要存放在内存存储介质中，如果需要快速地处理大规模数据，则需要提供高容量的内存；而 MapReduce 是面向磁盘的分布式计算框架，因此在成本考虑方面，Spark 的计算成本高于 MapReduce 计算框架。

（6）在简单易管理方面。目前 Spark 也在同一个集群上运行流处理、批处理和机器学习，同时 Spark 也可以管理不同类型的负载。这些都是 MapReduce 做不到的。

7.3 Spark 的工作机制

下面开始深入探讨 Spark 的内部工作原理，具体包括 Spark 运行的 DAG 图、Partition、容错机制、缓存管理以及数据持久化。

7.3.1 DAG 工作图

应用程序提交给 Spark 运行，通过生成 RDD DAG 的方式描述 Spark 应用程序的逻辑。

DAG 是有向无环图，是图论里面的概念，可以用图 G=< V，E >来描述，E 中的边都是有向边，顶点之间构成依赖关系，并且不能形成环路。当用户运行 action 操作的时候，Spark 调度器检查 RDD 的 lineage 图，生成一个 DAG 图，最后根据这个 DAG 图来分配任务执行。

为了 Spark 更加高效的调度和计算，RDD DAG 中还包括了宽依赖和窄依赖。窄依赖是父节点 RDD 中的分区最多只被子节点 RDD 中的一个分区使用；而宽依赖是父节点

RDD 中的分区被子节点 RDD 中的多个子分区使用，如图 7-1 所示。

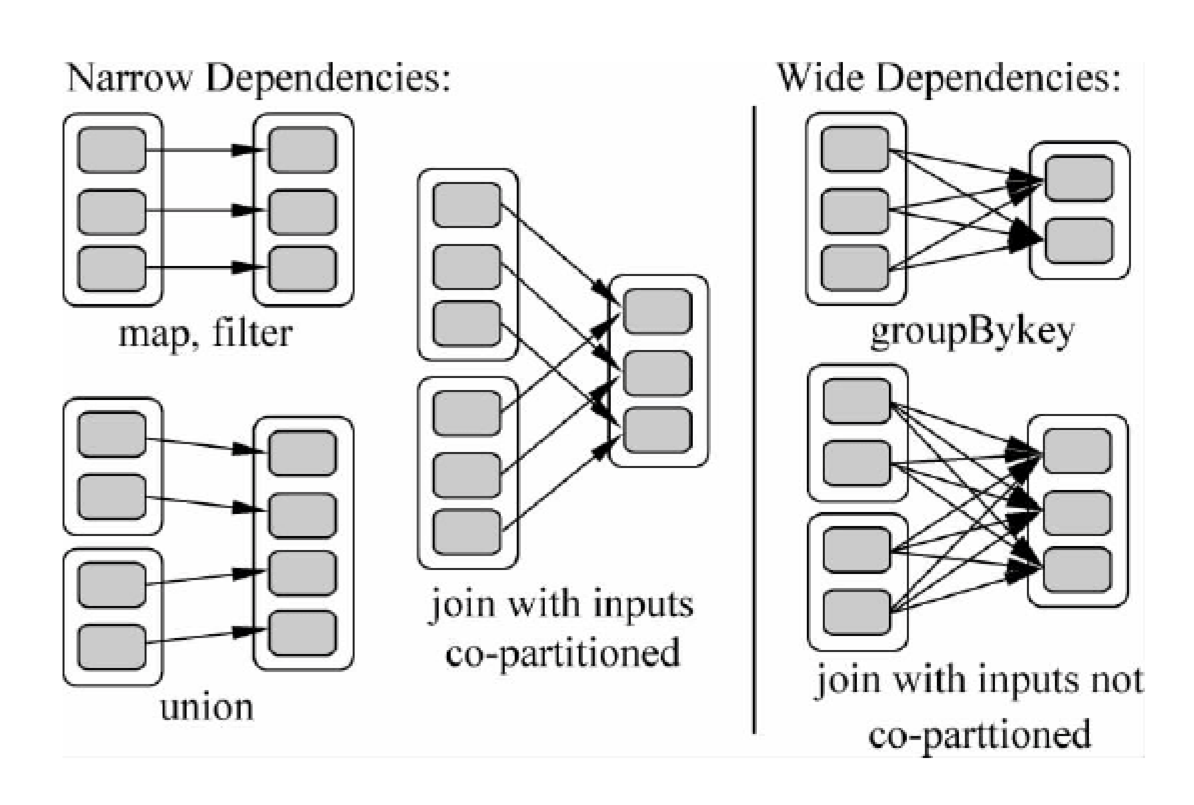


图 7-1　窄依赖和宽依赖

如图 7-1 所示，map 建立的 RDD 中的每个分区 Partition 只被子节点 filter RDD 中的一个子分区使用，所以是窄依赖；而 groupByKey 建立的 RDD 多个子分区 Partition 引用一个父节点 RDD 中的分区。

图 7-2 所示为 Spark 集群中一个应用程序的执行，生成了一个 DAG 图。

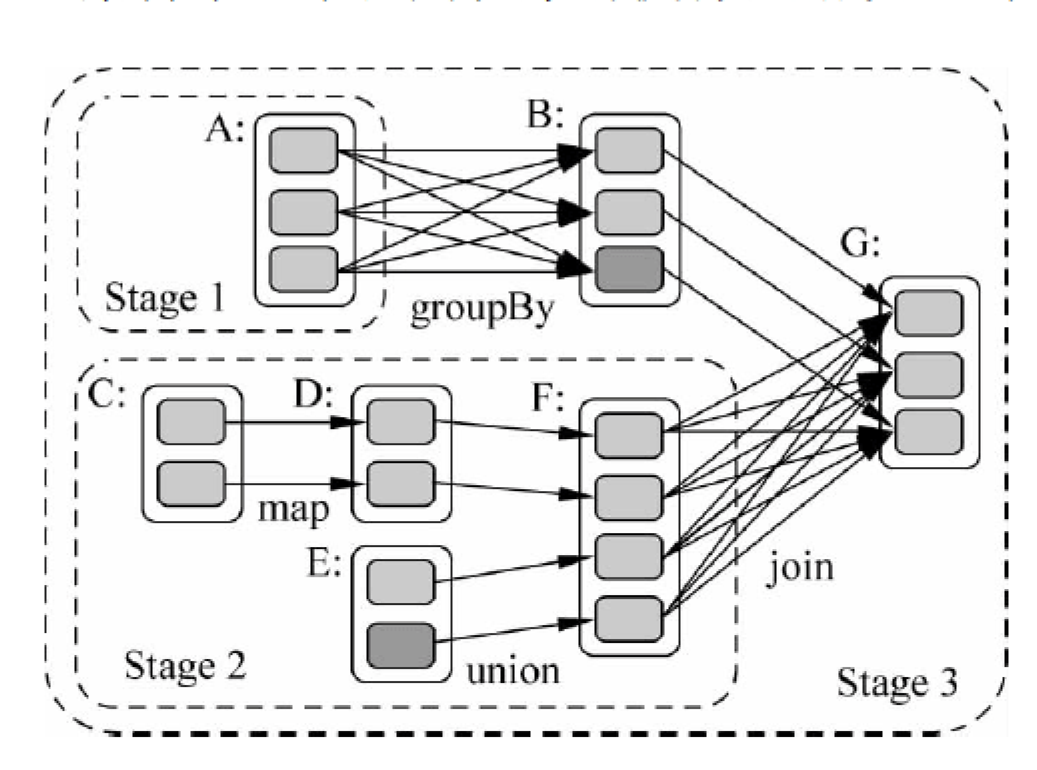


图 7-2　Spark 应用程序的执行

Spark 调度器根据 RDD 中的宽依赖和窄依赖形成 stage 的 DAG 图，如图 7-2 所示。每个 stage 是包含尽可能多的窄依赖的流水线 transformation。

采用 DAG 方式描述运行逻辑，可以描述更加复杂的运算功能，也有利于 Spark 调度器调度。

7.3.2　Partition

Spark 执行每次操作 transformation 都会产生一个新的 RDD，每个 RDD 是 Partition 分区的集合。在 Spark 中，操作的粒度是 Partition 分区，所有针对 RDD 的 map、filter 等

操作，最后都转换成对 Partition 的操作，每个 Partition 对应一个 Spark task。

当前支持的分区方式有 hash 分区和范围(range)分区。

7.3.3 Lineage 容错方法

在容错方面有多种方式，包括数据复制以及记录修改日志。但是由于 Spark 采用 DAG 描述 driver program 的运算逻辑，因此 Spark RDD 采用一种称为 Lineage 的容错方法。

RDD 本身是一个不可更改的数据集，Spark 根据 transformation 和 action 构建它的操作图 DAG，因此当执行任务的 Worker 失败时完全可以通过操作图 DAG 获得之前执行的操作，进行重新计算。由于无须采用 replication 方式支持容错，很好地降低了跨网络的数据传输成本。

不过，在某些场景下 Spark 也需要利用记录日志的方式来支持容错。针对 RDD 的 wide dependency，最有效的容错方式同样是采用 checkpoint 机制。当前，Spark 并没有引入 auto checkpointing 机制。

7.3.4 内存管理

旧版本 Spark(1.6 之前)的内存空间被分成了 3 块独立的区域，每块区域的内存容量是按照 JVM 堆大小的固定比例进行分配的：

- Execution。在执行 shuffle、join、sort 和 aggregation 时，Execution 用于缓存中间数据，通过 spark.shuffle.memoryFraction 进行配置，默认为 0.2。
- Storage。Storage 主要用于缓存数据块以提高性能，同时也用于连续不断地广播或发送大的任务结果，通过 spark.storage.memoryFraction 进行配置，默认为 0.6。
- Other。这部分内存用于存储运行 Spark 系统本身需要加载的代码与元数据，默认为 0.2。

无论是哪个区域的内存，只要内存的使用量达到了上限，则内存中存储的数据就会被放入到硬盘中，从而清理出足够的内存空间。这样，由于和执行或存储相关的数据在内存中不存在，就会影响到整个系统的性能，导致 I/O 增长，或者重复计算。

1. Execution 内存管理

Execution 内存进一步为多个运行在 JVM 中的任务分配内存。与整个内存分配的

方式不同，这块内存的再分配是动态分配的。在同一个 JVM 下，如果当前仅有一个任务正在执行，则它可以使用当前可用的所有 Execution 内存。

Spark 提供了以下 Manager 对这块内存进行管理：

- ShuffleMemoryManager。它扮演了一个中央决策者的角色，负责决定分配多少内存给哪些任务。一个 JVM 对应一个 ShuffleMemoryManager。
- TaskMemoryManager。它记录和管理每个任务的内存分配，实现为一个 page table，用于跟踪堆（heap）中的块，侦测异常抛出时可能导致的内存泄露。在其内部调用了 ExecutorMemoryManager 去执行实际的内存分配与内存释放。一个任务对应一个 TaskMemoryManager。
- ExecutorMemoryManager。其用于处理 on-heap 和 off-heap 的分配，实现为弱引用的池允许被释放的 page 可以被跨任务重用。一个 JVM 对应一个 ExecutorMemeoryManager。

内存管理的执行流程大致如下：

当一个任务需要分配一块大容量的内存用于存储数据时，首先会请求 ShuffleMemoryManager，告知“我想要 X 个字节的内存空间”。如果请求可以被满足，则任务就会要求 TaskMemoryManager 分配 X 个字节的空间。一旦 TaskMemoryManager 更新了它内部的 page table，就会要求 ExecutorMemoryManager 去执行内存空间的实际分配。

这里有一个内存分配的策略。假定当前的 active task 数据为 N，那么每个任务可以从 ShuffleMemoryManager 处获得多达 $1/N$ 的执行内存。分配内存的请求并不能完全得到保证，例如内存不足，这时任务就会将它自身的内存数据释放。根据操作的不同，任务可能重新发出请求，又或者尝试申请小一点的内存块。

2. Storage 的存储管理

Storage 内存由更加通用的 BlockManager 管理。如前所说，Storage 内存的主要功能是用于缓存 RDD Partitions，也用于将容量大的任务结果传播和发送给 driver。

Spark 提供了 Storage Level 来指定块的存放位置：Memory、Disk 或者 Off-Heap。Storage Level 还可以指定存储时是否按照序列化的格式。当 Storage Level 被设置为 MEMORY_AND_DISK_SER 时，内存中的数据以字节数组（byte array）形式存储，当这些数据被存储到硬盘中时，不再需要进行序列化。若设置为该 Level，则 evict 数据会更加高效。

到了 1.6 版本，Execution Memory 和 Storage Memory 之间支持跨界使用。当执行内存不够时可以借用存储内存，反之亦然。

7.3.5 数据持久化

Spark 最重要的一个功能是它可以通过各种操作(operation)持久化(或者缓存)一个集合到内存中。当用户持久化一个 RDD 的时候,每一个节点都将参与计算的所有分区数据存储到内存中,并且这些数据可以被这个集合(以及这个集合衍生的其他集合)的动作(action)重复利用。这个能力使后续的动作速度更快(通常快 10 倍以上)。对应迭代算法和快速的交互使用来说,缓存是一个关键的工具。

用户能通过 persist()或者 cache()方法持久化一个 RDD。首先在 action 中计算得到 RDD;然后将其保存在每个节点的内存中。Spark 的缓存是一个容错的技术,如果 RDD 的任何一个分区丢失,它可以通过原有的转换(transformation)操作自动地重复计算并且创建出这个分区。

此外,用户可以利用不同的存储级别存储每一个被持久化的 RDD。

7.4 数据的读取

Spark 支持多种外部数据源来创建 RDD,Hadoop 支持的所有格式 Spark 都支持。

7.4.1 HDFS

HDFS 是一个分布式文件系统,其目标就是运行在廉价的服务器上。HDFS 和 Hadoop MapReduce 构成了一整套的运行环境。Spark 可以很好的支持 HDFS。在 Spark 下要使用 HDFS 集群中的文件需要更改对应的配置文件,把 Hadoop 中的 hdfs-site.xml 和 core-site.xml 复制到 Spark 的 conf 目录下,这样就可以像使用普通的本地文件系统中的文件一样使用 HDFS 中的文件了。

7.4.2 Amazon S3

Amazon S3 提供了对象存储服务,目前使用广泛。Spark 提供了针对 S3 的文件输入服务支持。为了可以在 Spark 应用中读取和存储数据到 S3 中,可以使用 Hadoop 文件 API(SparkContext.hadoopFile、JavaHadoopRDD.saveAsHadoopFile、SparkContext.newAPIHadoopRDD 和 JavaHadoopRDD.saveAsNewAPIHadoopFile)来读和写 RDD。

用户可以采用以下方式来做 Word Count 应用：

```
scala> val sonnets = sc.textFile("s3a://s3-to-ec2/sonnets.txt")
scala> val counts = sonnets.flatMap(line => line.split(" ")).map(word =>(word, 1)).
reduceByKey(_ + _)
scala> counts.saveAsTextFile("s3a://s3-to-ec2/output")
```

7.4.3 HBase

HBase 是一个列数据库，一种 NoSQL，支持 CRUD 操作，具有容错、高可用、高可扩展以及高吞吐量等特点。Spark 也支持 HBase 的读取和写入操作。在采用 Spark 写入到 HBase 的过程中需要用到 PairRDDFunctions.saveAsHadoopDataset；在采用 Spark 读取 HBase 中的数据的时候需要用到使用 SparkContext 提供的 newAPIHadoopRDDAPI 将表的内容以 RDDs 的形式加载到 Spark 中。

7.5 应用案例

7.5.1 日志挖掘

采用 Spark 针对日志文件进行数据分析。根据 Tomcat 日志计算 URL 访问情况。区别于统计 GET 和 POST URL 访问量，其要求输出结果(访问方式、URL、访问量)。以下是简单的测试数据集样例：

```
196.168.2.1 - - [03/Jul/2014:23:57:42 +0800] "GET /html/notes/20140620/872.html
HTTP/1.0" 200 52373 0.034
196.168.2.1 - - [03/Jul/2014:23:58:17 +0800] "POST /service/notes/addViewTimes_900.
htm HTTP/1.0" 200 2 0.003
196.168.2.1 - - [03/Jul/2014:23:58:51 +0800] "GET /html/notes/20140617/888.html
HTTP/1.0" 200 70044 0.057
```

为了达到对应的日志分析结果，编写以下 Spark 代码：

```
//textFile() 加载数据
val data = sc.textFile("/spark/seven.txt")
```

```
//filter 过滤长度小于 0,过滤不包含 GET 与 POST 的 URL
val filtered = data.filter(_.length()>0).filter( line => (line.indexOf("GET")>0 ||
line.indexOf("POST")>0) )

//转换成键-值对的操作
val res = filtered.map( line =>{
if(line.indexOf("GET")>0){             //截取 GET 到 URL 的字符串
(line.substring(line.indexOf("GET"),line.indexOf("HTTP/1.0")).trim,1)
}else{                                 //截取 POST 到 URL 的字符串
(line.substring(line.indexOf("POST"),line.indexOf("HTTP/1.0")).trim,1)
}                                      //通过 reduceByKey 求 sum
}).reduceByKey(_+_)

//触发 action 事件执行
res.collect()
```

运行结果输出样例如下：

```
(POST /service/notes/addViewTimes_779.htm,1),
(GET /service/notes/addViewTimes_900.htm,1),
(POST /service/notes/addViewTimes_900.htm,1),
(GET /notes/index-top-3.htm,1),
(GET /html/notes/20140318/24.html,1),
(GET /html/notes/20140609/544.html,1),
(POST /service/notes/addViewTimes_542.htm,2)
```

7.5.2 判别西瓜好坏

西瓜是一种人们都很喜欢的水果，是盛夏季节的一种解暑物品。西瓜分为好瓜和坏瓜，我们都希望购买到的西瓜是好的。这里给出判断西瓜好坏的两个特征，一个特征是西瓜的糖度，另外一个特征是西瓜的密度，这两个数值都是 0～1 的小数。每个西瓜的好坏用数值来表示，1 表示好瓜，0 表示坏瓜。基于西瓜的测试数据集来判断西瓜的好坏。

Spark 中提供了 MLib 机器学习库，使用 MLib 机器学习库中提供的例子，采用 GBT 模型，训练参数，最后利用训练集测试 GBT 模型的好坏，判断西瓜的准确度。

详细的代码可以从 GitHub 上下载(https://github.com/alibook/alibook-bigdata.git),下面是利用 Spark GBT 模型的代码:

```
object SparkGBT {
  def main (args: Array[String]) {
    if (args.length < 0) {
      println("Usage:FilePath")
      sys.exit(1)
    }
    //初始化
    val conf = new SparkConf().setAppName("Spark MLlib Exercise: GradientBoostedTree")
    val sc = new SparkContext(conf)

    //数据文件加载和分析
    val data = MLUtils.loadLibSVMFile(sc, "/home/liujun/workplace/scala_GBT/GBT_data.txt")
    //数据拆分为训练集和测试集(30%测试)
    val splits = data.randomSplit(Array(0.7, 0.3))
    val (trainingData, testData) = (splits(0), splits(1))

    //训练 GBT 模型
    //默认情况下,defaultParams 分类使用 LogLoss
    val boostingStrategy = BoostingStrategy.defaultParams("Classification")
    boostingStrategy.numIterations = 10  //注意: 在实践中使用多个迭代
    boostingStrategy.treeStrategy.numClasses = 2
    boostingStrategy.treeStrategy.maxDepth = 3
    //空 categoricalFeaturesInfo 指示所有功能是连续的
    boostingStrategy.treeStrategy.categoricalFeaturesInfo = Map[Int, Int]()

    val model = GradientBoostedTrees.train(trainingData, boostingStrategy)

    //评估测试实例和试验误差的计算模型
    val labelAndPreds = testData.map { point =>
      val prediction = model.predict(point.features)
      (point.label, prediction)
    }
    val testErr = labelAndPreds.filter(r => r._1 != r._2).count.toDouble / testData.
count()
    println("Test Error = " + testErr)
    println("Learned classification GBT model:\n" + model.toDebugString)
```

```
    labelAndPreds.collect().foreach(x =>
      println("Lable and Prediction: " + x._1.toString + " " + x._2.toString))
  trainingData.saveAsTextFile("/home/liujun/workplace/scala_GBT/trainingData")
    testData.saveAsTextFile("/home/liujun/workplace/scala_GBT/testData")
  }
}
```

在终端上运行以下命令，在具体的环境中需要修改对应的文件路径名字：

```
build.sbt                    //设置好 sbt
sbt package exit             //运用 sbt 将文件打包
spark-2.0.0-bin-hadoop2.6/bin/spark-submit --master local --class
SparkClustering target/scala-2.11/sparkclustering_2.11-1.0.jar /home/liujun/
workplace/scala_Clustering/cluster
//最后提交到 Spark 集群上运行
```

测试结果及运行如图 7-3 和图 7-4 所示。

```
Test Error = 0.2
Learned classification GBT model:
TreeEnsembleModel classifier with 10 trees
```

图 7-3　GBT 测试结果

```
Lable and Prediction: 0.0 0.0
Lable and Prediction: 1.0 0.0
Lable and Prediction: 0.0 0.0
Lable and Prediction: [illegible].0 0.0
Lable and Prediction: 0.0 0.0
Lable and Prediction: 0.0 1.0
Lable and Prediction: 0.0 0.0
Lable and Prediction: 0.0 0.0
Lable and Prediction: 1.0 0.0
Lable and Prediction: 0.0 0.0
Lable and Prediction: 0.0 0.0
Lable and Prediction: 1.0 1.0
Lable and Prediction: 1.0 1.0
Lable and Prediction: 0.0 0.0
Lable and Prediction: 1.0 0.0
Lable and Prediction: 0.0 0.0
Lable and Prediction: 1.0 1.0
Lable and Prediction: 0.0 1.0
Lable and Prediction: 1.0 1.0
Lable and Prediction: 1.0 1.0
Lable and Prediction: 1.0 1.0
Lable and Prediction: 1.0 1.0
Lable and Prediction: 1.0 1.0
Lable and Prediction: 1.0 1.0
Lable and Prediction: 1.0 1.0
```

图 7-4　GBT 运行数据

7.6 Spark 的发展趋势

Spark 诞生于伯克利 AMP 实验室，起初是一个研究性质的项目，目标是为迭代式机器学习提供帮助。随着 Spark 的开源，因为其采用内存存储，计算速度比 MapReduce 更快，而且 Spark 简单、易用，受到了众多人的关注和喜爱。目前 Apache Spark 社区非常活跃，并且以 Spark RDD 为核心，逐步形成了 Spark 的生态圈，包括 Spark SQL、Spark Streaming、Spark MLib 等众多上层数据分析工具以及实时处理框架。

目前 Spark 已经在国内外各大公司使用，包括 eBay、Yahoo!、IBM、阿里、百度、腾讯等众多公司。实践表明 Spark 性能优越，各大公司在 Spark 上的投入也比较大，因此 Spark 生态也在不断完善，不断有新的 Spark 生态圈中的框架出现，包括 Tachyon 分布式内存文件系统、SparkR 统计框架。

7.7 习题

1. 什么是 Spark RDD? 简要介绍 RDD 的创建方法。
2. 什么是 DAG? Spark 的 DAG 如何生成?
3. 简述 Spark RDD 的容错方法。
4. 简述 Spark 的内存管理的工作原理。
5. 什么是 Spark 的分区 Partition?

第 8 章

流 计 算

8.1 流计算概述

在传统的数据处理流程中总是先收集数据，然后将数据放到 DB 中。当人们需要的时候通过 DB 对数据做 query，得到答案或进行相关的处理。这样看起来虽然非常合理，但是结果却不理想，尤其是对一些实时搜索应用环境中的某些具体问题，采用类似于 MapReduce 方式的离线处理并不能很好地解决问题，这就引出了一种新的数据计算结构——流计算方式。它可以很好地对大规模流动数据在不断变化的运动过程中实时地进行分析，捕捉到可能有用的信息，并把结果发送到下一计算节点。

比较早期的代表系统有 IBM 公司的 System S，它是一个完整的计算架构，通过"stream computing"技术可以对 stream 形式的数据进行 real-time 的分析。最初的系统拥有大约 800 个微处理器，但 IBM 称，根据需求，这个数字也有可能上万。研究者讲到，其中最关键的部分是 System S 软件，它可以将任务分开，比如分为图像识别和文本识别，然后将处理后的结果碎片组成完整的答案。IBM 实验室的高性能流运算项目的负责人 Nagui Halim 谈到：System S 是一个全新的运算模式，它的灵活性和速度颇具优势。与传统系统相比，它的方式更加智能化，可以适当转变，以适用于需要解决的问题。

目前流式计算是业界研究的一个热点，最近 Twitter、LinkedIn 等公司相继开源了流式计算系统 Storm、Kafka 等，Twitter 最近又公布了新的流式计算框架 Hron，加上 Yahoo！之前开源的 S4，流式计算研究在互联网领域持续升温。不过，流式计算并非是最近几年才开始研究，传统行业（像金融领域等）很早就已经在使用流式计算系统，比较

知名的有 StreamBase、Borealis 等。

8.2 流计算与批处理系统的对比

流计算侧重于实时计算方面，而批处理系统侧重于离线数据处理方面；一个追求的是低延迟，另外一个追求的是高吞吐量；处理的数据也不同，流计算处理的数据经常不断变化，而离线处理的数据是静态数据，输出形式也不同。总体来讲，两者的区别体现在以下几个方面。

(1) 系统的输入包括两类数据，即实时的流式数据和静态的离线数据。其中，流式数据是前端设备实时发送的识别数据、GPS 数据等，是通过消息中间件实现的事件触发推送至系统的。离线数据是应用需要用到的基础数据(提前梳理好的)等关系数据库中的离线数据，是通过数据库读取接口获取而批量处理的系统。

(2) 系统的输出也包括流式数据和离线数据。其中，流式数据是写入消息中间件的指定数据队列缓存，可以被异步推送至其他业务系统。离线数据是计算结果，直接通过接口写入业务系统的关系型数据库。

(3) 业务的计算结果输出方式是通过两个条件决定的。一是结果产生的频率。若计算结果产生的频率可能会较高，则结果以流式数据的形式写入消息中间件(比如要实时监控该客户所拥有的标签，也就是说要以极高的速度被返回，这类结果以流式数据形式被写入消息中间件)。这是因为数据库的吞吐量很可能无法适应高速数据的存取需求。二是结果需要写入的数据库表规模。若需要插入结果的数据表已经很庞大，则结果以流式数据的形式写入消息中间件，待应用层程序实现相关队列数据的定期或定量的批量数据库转储(比如宽表异常庞大，每次查询数据库都会有很高的延迟，那么就将结果信息暂时存入中间件层，在晚些时候再定时或定量地进行批量数据库转储)。这是因为大数据表的读取和写入操作对毫秒级别的响应时间仍然无能为力。若对以上两个条件均无要求，结果可以直接写入数据库的相应表中。

8.3 Storm 流计算系统

Storm 是一个 Twitter 开源的分布式、高容错的实时计算系统。Storm 令持续不断的流计算变得容易，弥补了 Hadoop 批处理不能满足的实时要求。Storm 经常用于实时分析、在线机器学习、持续计算、分布式远程调用和 ETL 等领域。Storm 的部署管理非常

简单，而且在同类的流式计算工具中 Storm 的性能也是非常出众的。

Storm 主要分为 Nimbus 和 Supervisor 两种组件，集群架构如图 8-2 所示。这两种组件都是快速失败的，没有状态。任务状态和心跳信息等都保存在 ZooKeeper 上，提交的代码资源都在本地机器的硬盘上。

(1) Nimbus 负责在集群里面发送代码，分配工作给机器，并且监控状态。全局只有一个。

(2) Supervisor 会监听分配给它那台机器的工作，根据需要启动/关闭工作进程 Worker。每一个要运行 Storm 的机器上都要部署一个，并且按照机器的配置设定上面分配的槽位数。

(3) ZooKeeper 是 Storm 重点依赖的外部资源。Nimbus 和 Supervisor 甚至实际运行的 Worker 都是把心跳信息保存在 ZooKeeper 上。Nimbus 也是根据 ZooKeerper 上的心跳信息和任务运行状况进行调度和任务分配的。

(4) Storm 提交运行的程序称为 Topology。

(5) Topology 处理的最小消息单位是一个 Tuple，也就是一个任意对象的数组。

(6) Topology 由 Spout 和 Bolt 构成。Spout 是发出 Tuple 的节点。Bolt 可以随意订阅某个 Spout 或者 Bolt 发出的 Tuple。Spout 和 Bolt 统称为 Component。

图 8-1 是一个 Topology 设计的逻辑视图，图 8-2 是 Storm 集群架构。

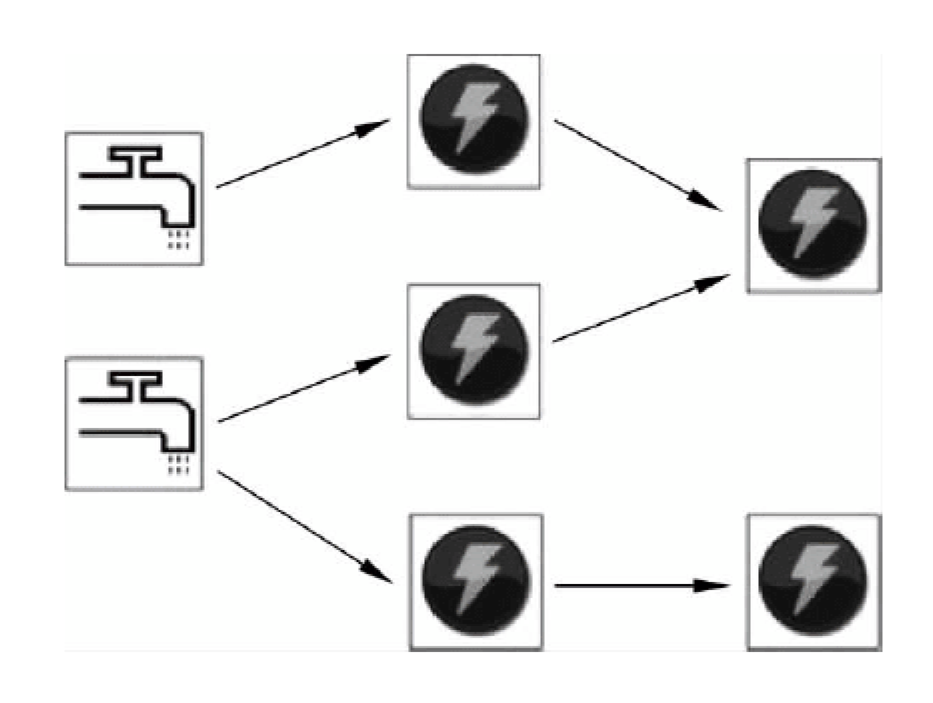


图 8-1 Topology 设计的逻辑图

图 8-3 所示为 Storm 工作流。

整体的 Storm 工作流步骤如下。

(1) 在初始情况下，Nimbus 等待客户端提交 Storm Topology。

(2) 一旦一个 Topology 提交后，Nimbus 将会处理这个 Topology，安排将要执行的所有任务。

(3) 一旦所有的工作节点的信息都收集完成，Nimbus 将分发所有的任务到各个计算节点上。

(4) 在一定的时间间隔内，所有的 Supervisor 都会发送心跳信息给 Nimbus，告诉

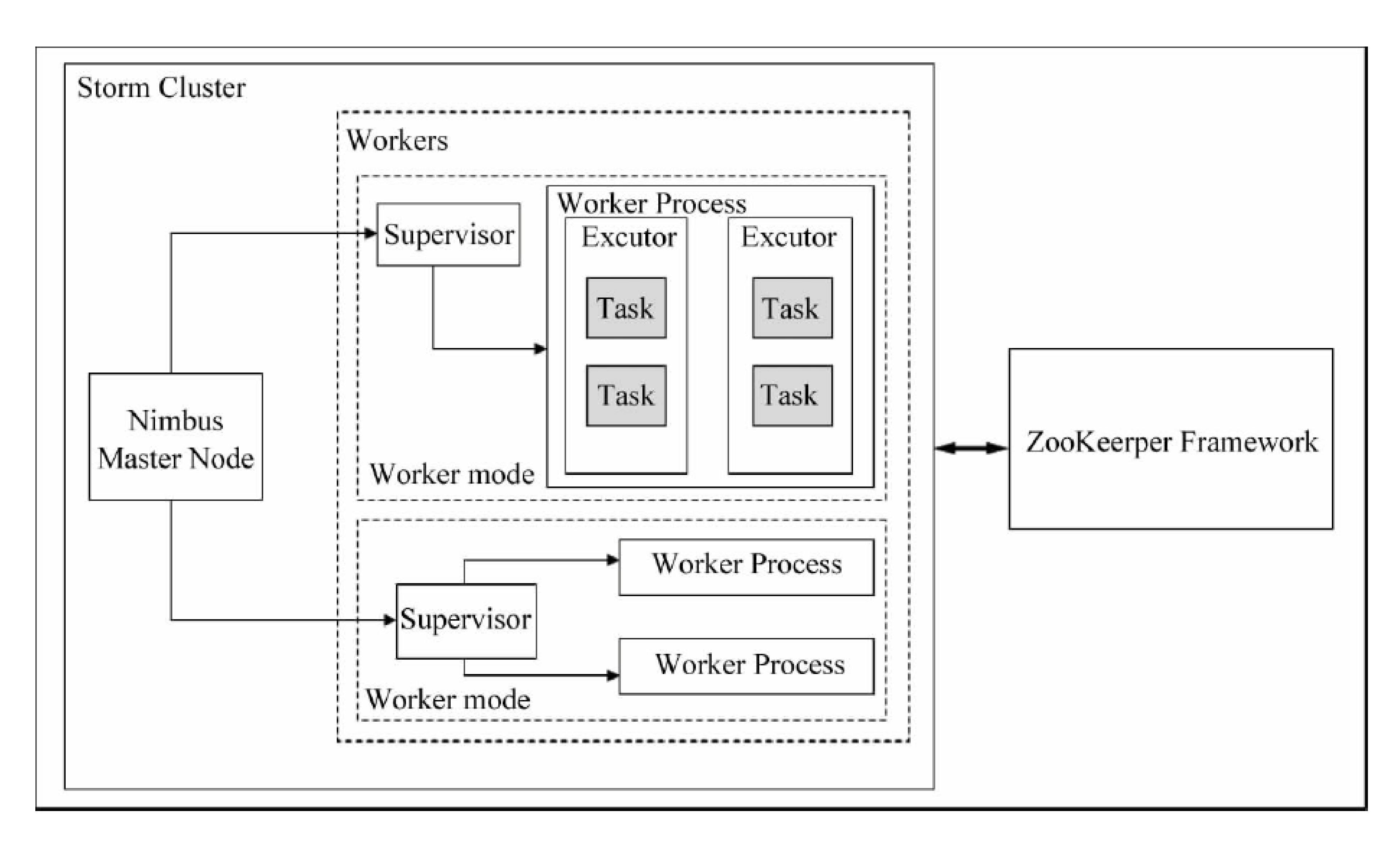


图 8-2 Storm 集群架构

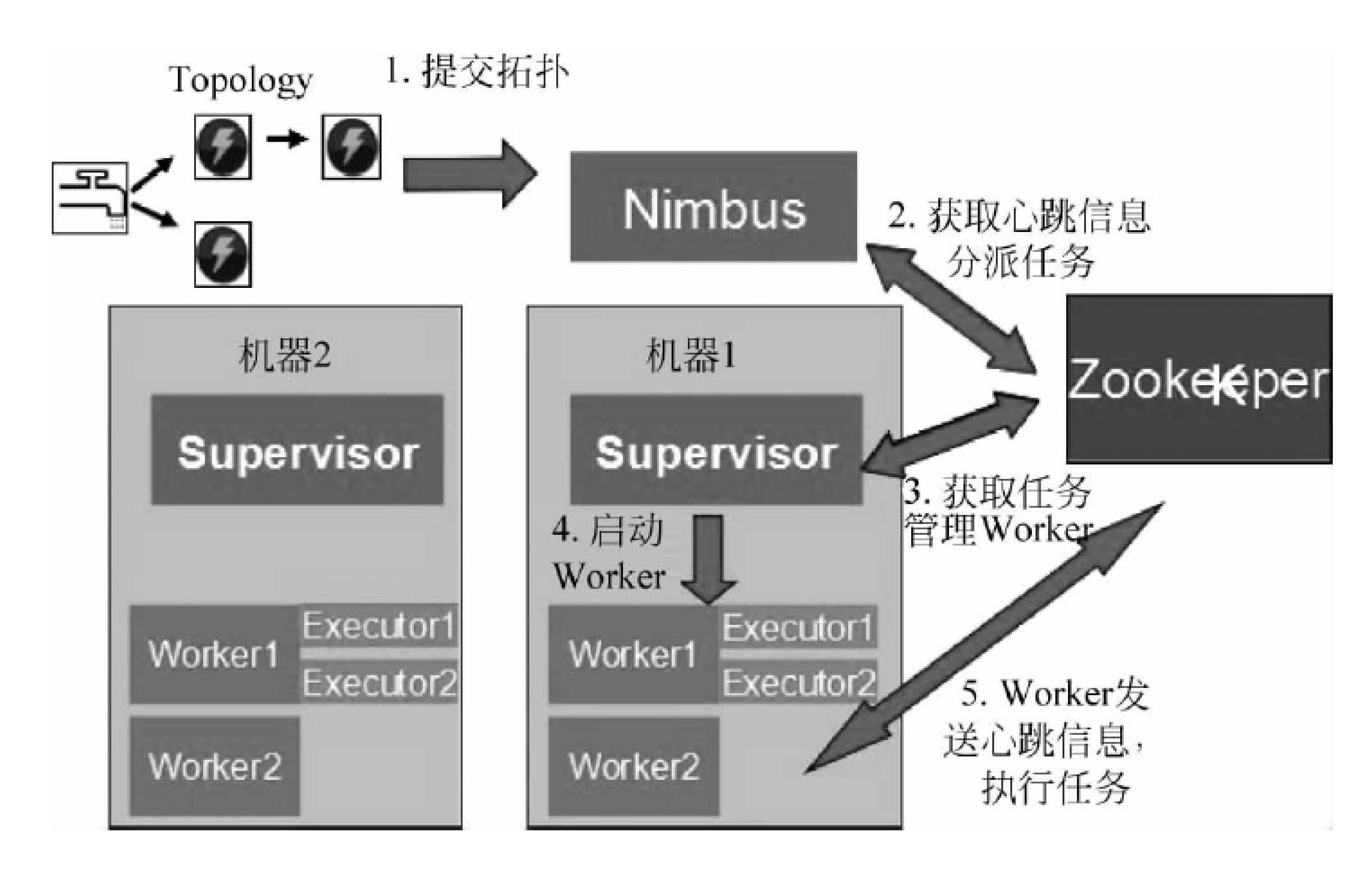


图 8-3 Storm 工作流

Nimbus 该 Supervisor 正常运行。

(5) 当 Supervisor 失效时，没有发送心跳信息给 Nimbus，此时 Nimbus 会把任务赋给其他 Supervisor。

(6) 当 Nimbus 失效的时候，Supervisor 会正常运行以前赋给该 Supervisor 的任务。

(7) 一旦所有的任务都完成，Supervisor 会等一个新的任务发送过来。

(8) 重新启动的 Nimbus 从它失效的那个地方继续启动。类似地，重新启动的 Supervisor 也是从它停止的地方继续启动。Storm 确保所有的任务至少执行一次。

(9) 当所有 Topology 都完成的时候，Nimbus 等待新的 Topology 到达；同理，Supervisor 也是类似。

8.4 Samza 流计算系统

Apache Samza 是一个分布式流处理框架。它使用 Apache Kafka 用于消息发送，采用 Apache Hadoop YARN 来提供容错、处理器隔离、安全性和资源管理，专用于实时数据的处理，非常像 Twitter 的流处理系统 Storm。Samza 非常适用于实时流数据处理的业务（如同 Apache Storm），如数据跟踪、日志服务、实时服务等应用，它能够帮助开发者进行高速消息处理，同时还具有良好的容错能力。在 Samza 流数据处理过程中，每个 Kafka 集群都与一个能运行 YARN 的集群相连并处理 Samza 作业。

Samza 由以下 3 层构成：

- 数据流层（A streaming layer）；
- 执行层（An execution layer）；
- 处理层（A progressing layer）。

整体的 Samza 架构通过图 8-4 所示的 3 个模块完成。

- 数据流：分布式消息中间件 Kafka。
- 执行：Hadoop 资源调度管理系统 YARN。
- 处理：Samza API。

Samza 通过使用 YARN 和 Kafka 提供一个阶段性的流处理和分区的框架，如图 8-5 所示。

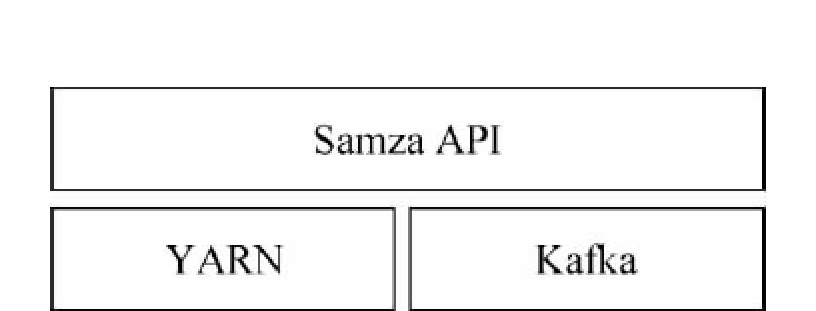


图 8-4 Samza 的功能模块

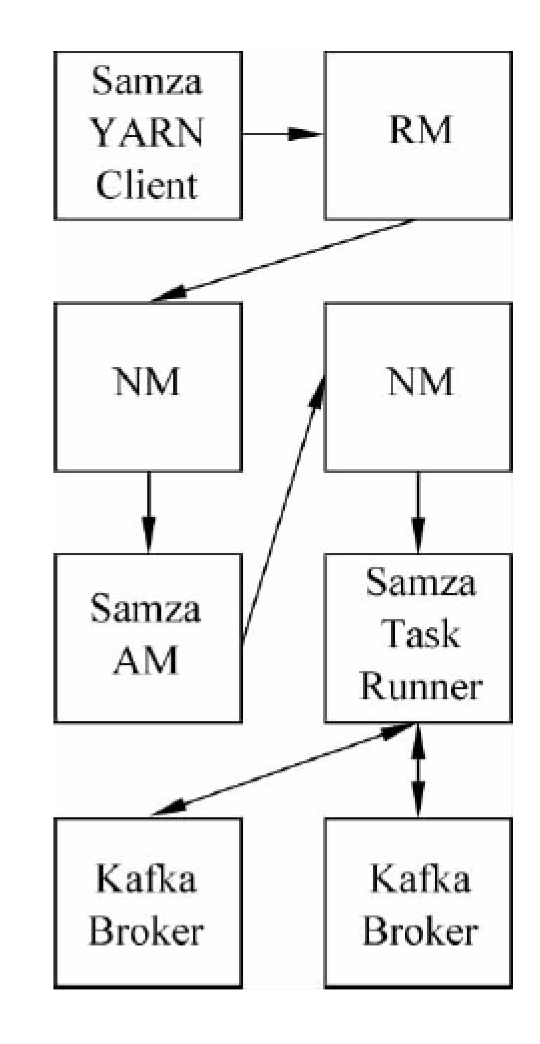


图 8-5 Samza、YARN 和 Kafka 模块之间的互动

Samza 的客户端使用 YARN 来运行一个 Samza 任务(Job)：YARN 启动并且监控一个或者多个 Samza Container，同时用户的处理逻辑代码(使用 StreamTask API)在这些容器里运行。这些 Samza 流任务的输入和输出都来自 Kafka 的 Broker(通常它们是作为 YARN NM 位于同台机器)。

进一步来说，第一个任务是分组工作通过将带有相同 userid 的消息发送到一个中间话题的相同分区里，用户可以通过使用第一个 Job 发射的消息里的 userid 作为 key 来实现，并且这个 key 被映射到这个中间话题的分区(通常会取 key 对分区数目取余)。第二个任务处理中间话题产生的消息。在第二个任务里每个任务都会处理中间话题的一个分区。在对应分区中任务会针对每一个 userid 做一个计数器，并且每次任务接收带着一个特定 userid 的消息时对应的计数器自增 1。

Kafka 接收到第一个 Job 发送的消息把它们缓冲到硬盘，并且分布在多台机器上。这有助于系统的容错性提升：如果一台机器挂了，没有消息会被丢失，因为它们被存在其他机器里。并且如果第二个 Job 因为某些原因消费消息的速度慢下来或者停止，第一个任务也不会受到影响：磁盘缓冲可以积累消息直到第二个任务快起来。

通过对 topic 分区，将数据流处理拆解到任务中以及在多台机器上并行执行任务，使得 Samza 具有很高的消息吞吐量。通过结合 YARN 和 Kafka，Samza 实现了高容错：如果一个进程或者机器失败，它会自动在另一台机器上重启并且继续从消息终端的地方开始处理，这些都是自动化的。

8.5 阿里云流计算

Aliyun StreamCompute(阿里云流计算)是运行在阿里云平台上的流式大数据分析平台，给用户提供在云上进行流式数据实时化分析的工具。使用阿里云 StreamSQL，用户可以轻松地搭建自己的流式数据分析和计算服务，彻底规避掉底层流式处理逻辑的繁杂的重复开发工作。

阿里云流计算提供类标准的 StreamSQL 语义协助用户简单、轻松地完成流式计算逻辑的处理。同时受限于 SQL 代码功能，无法满足某些特定场景的业务需求，阿里云流计算同时为部分授信用户提供全功能的 UDF 函数，帮助用户完成业务定制化的数据处理逻辑。在流数据分析领域，用户直接使用 StreamSQL＋UDF 即可完成大部分流式数据分析处理逻辑；同样受限于 SQL 的表达能力，目前的流计算更擅长于做流式数据分析、统计、处理，对于非 SQL 能够解决的领域，例如复杂的迭代数据处理、复杂的规则引擎告警则不适合用现有的流计算产品去解决。

目前，流计算擅长解决以下几个领域的应用场景问题。

(1) 实时的网络点击 PV、UV 统计。

(2) 统计交通卡口的平均 5 分钟通过的车流量。

(3) 水利大坝的压力数据统计和展现。

(4) 网络支付涉及金融盗窃固定行为规则的告警。

下面简单介绍阿里云流计算的系统架构情况，图 8-6 是阿里云流计算的处理流程。

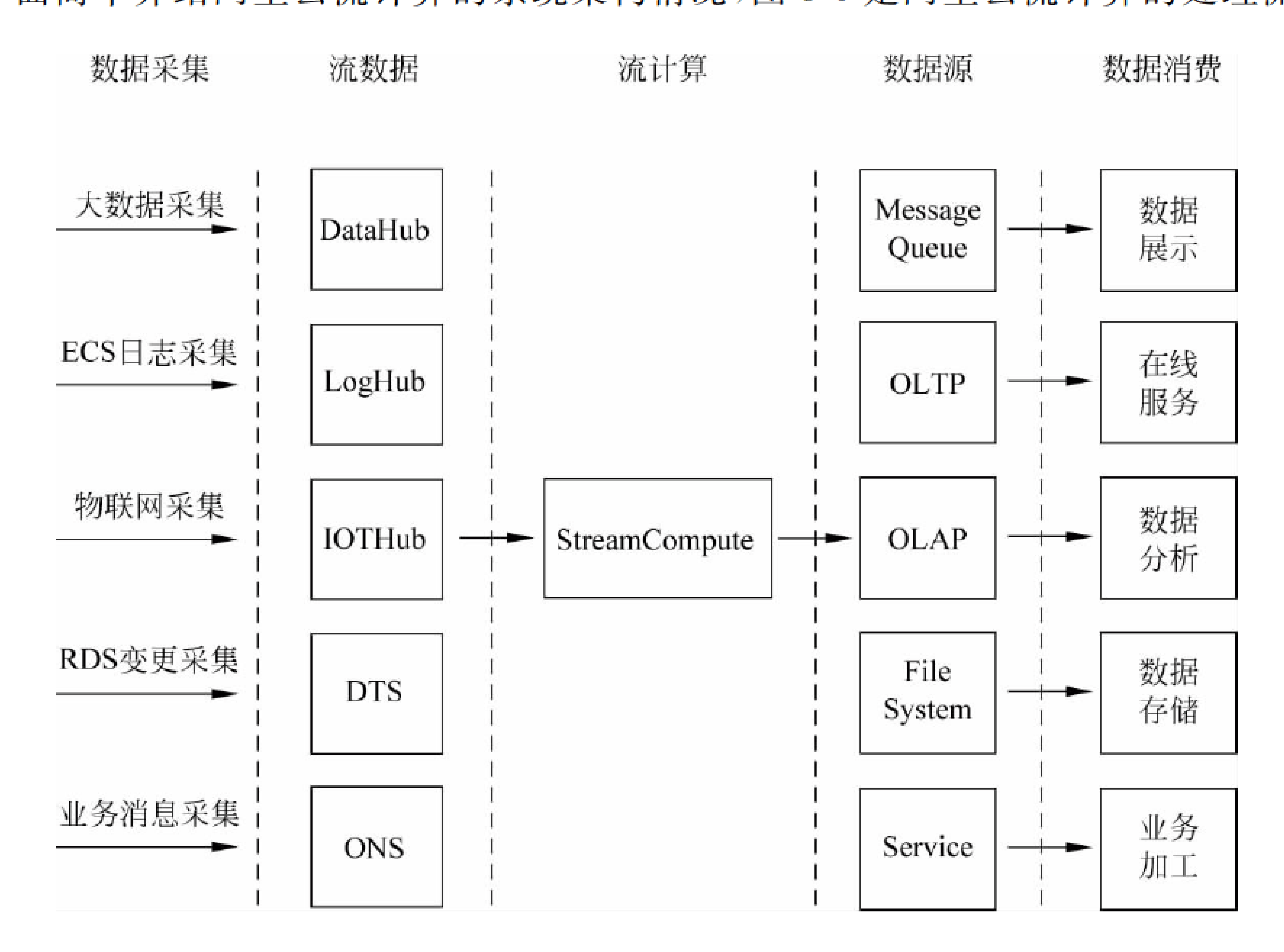


图 8-6 阿里云流计算的处理流程

1. 数据采集

广义的实时数据采集指用户使用流式数据采集工具将数据流式且实时地采集并传输到大数据 Pub/Sub 系统，该系统将为下游流计算提供源源不断的事件源去触发流式计算任务的运行。阿里云大数据生态中提供了诸多针对不同场景领域的流式数据 Pub/Sub 系统，阿里云流计算天然集成图 8-6 中诸多的 Pub/Sub 系统，以方便用户可以轻松地集成各类流式数据存储系统。例如用户可以直接使用流计算对接 SLS 的 LogHub 系统，以做到快速集成并使用 ECS 日志。

2. 流计算

流数据作为流计算的触发源驱动流计算运行，因此，一个流计算任务必须至少使用一个流数据作为数据源。对于一些业务较复杂的场景，流计算还支持和静态数据存储进行关联查询。例如针对每条 DataHub 流式数据，流计算将根据流式数据的主键和 RDS

中的数据进行关联查询(即 join 查询);同时,阿里云流计算还支持针对多条数据流进行关联操作,StreamSQL 支持阿里集团量级的复杂业务也不在话下。

3. 实时数据集成

为尽可能减少数据处理的时延,同时减少数据链路的复杂度,阿里云流计算可将计算的结果数据不经其他过程直接写入目的数据源,从而最大程度地降低全链路数据时延,保证数据加工的新鲜度。为了打通阿里云生态,阿里云流计算天然集成了 OLTP (RDS 产品线等)、NoSQL(OTS 等)、OLAP(ADS 等)、MessageQueue(DataHub、ONS 等)、MassiveStorage(OSS、MaxCompute 等)。

4. 数据消费

流计算的结果数据进入各类数据源后,用户可以使用各类个性化的应用消费结果数据。例如用户可以使用数据存储系统访问数据,使用消息投递系统进行信息接收,或者直接使用告警系统进行告警。

每年的天猫双十一购物狂欢节已逐渐成为全球互联网最大规模的商业促销狂欢活动。而每年的双十一节除了"买买买"之外,最引人注目的就是天猫双十一大屏不停闪变跳跃的总体成交总额。这份实时化的大数据展示链路凝结了阿里集团诸多顶尖级工程师长达数月的辛勤劳作,其中关键指标也颇具亮点,包括从天猫交易下单购买到数据采集、数据计算、数据校验,最终落到双十一大屏上展现的全链路时间压缩在 5 秒以内,0 点顶峰计算性能高达数十万订单/秒,多条链路流计算备份确保万无一失。

8.6 集群日志文件的实时分析

流计算适用于大规模实时计算分析,使用的生产环境包括股票市场分析、证券、传感器数据分析等,也可以用于实时分析当前系统的运行状态。目前分布式系统在各大生产系统中广泛使用,监控这些分布式系统产生的日志,进而分析这些系统的运行状态,判断集群运行是否正常,采用流计算框架实时分析分布式系统产生的日志。

下面以分析 HDFS 集群运行状态来简单说明流式计算框架的使用。HDFS 集群由 3 个部分组成,即 NameNode、DataNode 和 SecondaryNameNode。NameNode 保存所有的元数据信息,以及管理所有的 DataNode,一个健康和正常运行的 NameNode 节点对于一个正常的 HDFS 集群至关重要,当 NameNode 出现故障的时候需要及时报警,从而最大程度地减少损失。分析一个 NameNode 节点是否运行正常,一个重要的方法就是查看

NameNode 的日志文件，当 NameNode 的运行出现不正常情况时会产生 WARN 和 ERROR 日志信息。

下面利用 Flink 做简单的日志文件单词统计分析，分析一个时间段内 NameNode 产生的单词统计。将 HDFS 集群的 NameNode 产生的日志文件重定向到 netcat 命令，生成一个文件服务器。Flink 流应用程序接受来自 netcat 端的文本数据，然后统计单词个数，最后在一个具体的 Flink 集群节点上生成运算结果并显示。

Flink 基于网络文本数据的实时单词统计分析代码，详细的可运行代码可以从 GitHub 上下载(https://github.com/alibook/alibook-bigdata.git)，以下是部分代码。

```
public class SocketTextStream {
    public static void main(String[] args) throws Exception {

        if (!parseParameters(args)) {
            return;
        }

        //建立一个执行环境
        final StreamExecutionEnvironment env = StreamExecutionEnvironment
                .getExecutionEnvironment();

        //获取输入数据
        DataStream<String> text = env.socketTextStream(hostName, port, '\n', 0);

        DataStream<Tuple2<String, Integer>> counts =

        //拆分成对的线(二元组)包含: (word,1)
        text.flatMap(new Tokenizer())
        //由元组字段"0"分组并且合计元组字段"1"
            .keyBy(0)
            .sum(1);

        if (fileOutput) {
            counts.writeAsText(outputPath, WriteMode.NO_OVERWRITE);
        } else {
            counts.print();
        }

        //执行程序
```

```
        env.execute("WordCount from SocketTextStream Example");
    }

    // *************************************************************
    // UTIL METHODS
    // *************************************************************

    private static boolean fileOutput = false;
    private static String hostName;
    private static int port;
    private static String outputPath;

    private static boolean parseParameters(String[] args) {

        //输入参数分析
        if (args.length == 3) {
            fileOutput = true;
            hostName = args[0];
            port = Integer.valueOf(args[1]);
            outputPath = args[2];
        } else if (args.length == 2) {
            hostName = args[0];
            port = Integer.valueOf(args[1]);
        } else {
            System.err.println("Usage: SocketTextStreamWordCount <hostname><port>
                [<output path>]");
            return false;
        }
        return true;
    }


    /**
      * Implements the string tokenizer that splits sentences into words * as auser -
defined FlatMapFunction. The function takes a line * (String) andsplits it into multiple
pairs in the form of "(word,1)" * ({@code Tuple2<String,Integer>}).
      */
    public static final class Tokenizer implements FlatMapFunction < String, Tuple2
<String, Integer>> {
```

```
            private static final long serialVersionUID = 1L;

            public void flatMap(String value, Collector<Tuple2<String, Integer>> out)
                    throws Exception {
                //规范和分割线
                String[] tokens = value.toLowerCase().split("\\W+");

                //发出对
                for (String token : tokens) {
                    if (token.length() > 0) {
                        out.collect(new Tuple2<String, Integer>(token, 1));
                    }
                }
            }
        }

    }
```

首先在终端上生成文件服务器，在 HDFS 集群的 NameNode 节点的终端上运行以下命令。

```
tail -f hadoop-qzhong-namenode-nobida144.log | nc -l 12345
```

上面的 hadoop-qzhong-namenode-nobida144.log 为 HDFS 集群 NameNode 产生的日志文件，12345 为网络文件传输的端口号。

然后将利用 Maven 编译好的 JAR 文件在 Flink 上运行，在 Flink 集群节点上运行以下命令。

```
bin/flink run -c alibook.flink.SocketTextStream /home/qzhong/flink-0.0.1.jar
nobida144flink 12345
```

运行结果如图 8-7 所示。

```
11/21/2016 11:11:17     Job execution switched to status RUNNING.
11/21/2016 11:11:17     Source: Socket Stream -> Flat Map(1/1) switched to SCHEDULED
11/21/2016 11:11:17     Source: Socket Stream -> Flat Map(1/1) switched to DEPLOYING
11/21/2016 11:11:17     Keyed Aggregation -> Sink: Unnamed(1/1) switched to SCHEDULED
11/21/2016 11:11:17     Keyed Aggregation -> Sink: Unnamed(1/1) switched to DEPLOYING
11/21/2016 11:11:17     Keyed Aggregation -> Sink: Unnamed(1/1) switched to RUNNING
11/21/2016 11:11:17     Source: Socket Stream -> Flat Map(1/1) switched to RUNNING
```

图 8-7 SocketTextStream 任务启动

然后根据 Flink 的 Web 界面查看 SocketTextStream 任务，找到对应的 Flink 文本统计计算节点，如图 8-8 所示。

Subtasks　TaskManagers　Accumulators　Checkpoints　Back Pressure

Start Time	End Time	Duration	Name	Bytes received	Records received	Bytes sent	Records sent	Tasks	Status
2016-11-22, 0:11:17	2016-11-22, 0:15:12	3m 55s	Source: Socket Stream -> Flat Map	0 B	0	482 KB	49,721	0 0 1 0 0 0 0	RUNNING

Start Time	End Time	Duration	Bytes received	Records received	Bytes sent	Records sent	Attempt	Host	Status
2016-11-22, 0:11:17		3m 55s	0 B	0	482 KB	49,721	1	nobida148:13524	RUNNING

Start Time	End Time	Duration	Name	Bytes received	Records received	Bytes sent	Records sent	Tasks	Status
2016-11-22, 0:11:17	2016-11-22, 0:15:12	3m 55s	Keyed Aggregation -> Sink: Unnamed	482 KB	49,721	0 B	0	0 0 1 0 0 0 0	RUNNING

图 8-8　用 Flink 查看 SocketTextStream 任务

接下来在节点 nobida148 上查看具体的任务单词统计情况，运行结果如图 8-9 所示。

```
[qzhong@nobida148 log]$ tail -f flink-qzhong-taskmanager-2-nobida148.out
(1,318)
(148,75)
(39402,75)
(call,292)
(2215,12)
(retry,219)
(0,419)
(wrote,73)
(41,73)
(bytes,73)
(2016,468)
(11,935)
(21,468)
(11,936)
(20,38)
(04,40)
(808,2)
(debug,468)
(namenode,15)
(namenoderesourcechecker,29)
(namenoderesourcechecker,30)
(java,541)
(isresourceavailable,15)
```

图 8-9　SocketTextStream 单词统计结果

8.7　流计算的发展趋势

本节分别从流计算技术发展和流计算应用趋势两个方面阐述。

在流计算技术发展方面，随着互联网技术的不断发展，互联网产生的数据不断增加，

传统的离线处理方式无法适用于不断变化的数据以及无法满足数据分析的低延迟要求，流计算框架可以很好地适应不断变化的数据以及实时处理数据。为了满足流计算的实时特性，目前流计算框架基本上把大规模数据存放在内存中，如 Spark Streaming、Flink 等；主要是目前内存存储容量不断增加，单位存储成本不断降低，内存存取访问速度在纳秒级别。因此建立以内存为基础的实时计算框架是流计算的一个发展趋势。

作为一个通用的计算框架，流计算框架也必须提供容错机制，提高系统可靠性。流计算框架应该提供一种更好的容错机制，传统的批处理采用的是以重做的方式来提供容错功能，但是该方式适合于短任务的执行，并不能很好地适用于流计算框架。因此流计算框架在容错性方面需要提供更短的时间以恢复错误的计算任务。

在流计算应用趋势方面，目前流计算框架在股票分析、传感器数据分析、智能交通数据分析等领域不断发展，同时也在在线学习方面不断取得进步，并不断扩展到其他实时分析领域。

8.8 习题

1. 简述流计算和批处理系统的区别。
2. 简述 Storm 流计算框架的架构以及 Storm 集群工作流状态。
3. 简述 Samza 流计算框架的架构、运行工作原理。
4. 动手构建一个关于天气的实时预警分析应用，采用 Storm 流计算框架。

第 9 章

图 计 算

9.1 图计算概述

在现实生活中人们会遇到大量采用图作为数据模型的计算问题，这类计算问题以顶点为中心，典型的如社交网络分析、移动电话网络分析、已发表科学成果之间的相互引用关系计算等，这类问题统称为图计算问题。解决图计算问题的一般方法有最短路径计算、聚类以及 PageRank 的变体。

就目前而言，需要处理的图计算问题的规模正在快速增大并且变得更加复杂，涉及大量的边和顶点，因此工业界和学术界都在为高效解决图计算问题而不断努力，已经开发出很多分布式图计算框架系统。以 Google Pregel 为代表的分布式图计算框架采用 BSP 计算模型处理 Google 公司内部的大量图任务。Apache 软件基金会根据 Google 发表的 Pregel 论文实现了一个开源的分布式图计算框架 Giraph。不同于 Pregel 和 Giraph 采用消息传递机制作为计算节点之间的通信方式，卡内基梅隆大学开发了分布式图计算框架 GraphLab，采用 GAS(Gather、Apply、Scatter)拉数据(data-pulling)模型和共享内存抽象，图计算算法开发人员只需要为每个顶点实现用户定义的 GAS 函数即可。

简而言之，分布式图框架就是将大型图的各种操作封装成接口，让分布式存储、并行计算等复杂问题对上层透明，从而使工程师将焦点放在图相关的模型设计和使用上，而不用关心底层的实现细节。分布式图框架的实现需要考虑两个问题，一是怎样切分图以更好地计算和保存；二是采用什么图计算模型。

1. 图切分方式

图的切分从总体上说有点切分和边切分两种方式。

（1）点切分。通过点切分之后，每条边只保存一次，并且出现在同一台机器上。邻居多的点会被分发到不同的节点上，增加了存储空间，并且有可能产生同步问题。它的优点是减少了网络通信。

（2）边切分。通过边切分之后，顶点只保存一次，切断的边打断后会保存在两台机器上。在进行基于边的操作时，对于两个顶点分到两个不同的机器的边来说，需要进行网络传输数据。这增加了网络传输的数据量，但好处是节约了存储空间。

2. BSP 计算模型

在 BSP 模型中，一次计算过程由一系列全局超步组成，每一个超步由并发计算、通信和同步 3 个步骤组成。同步完成标志着这个超步的完成及下一个超步的开始。BSP 模型的准则是批量同步（bulk synchrony），其独特之处在于超步（superstep）概念的引入。一个 BSP 程序同时具有水平和垂直两个方面的结构。从垂直上看，一个 BSP 程序由一系列串行的超步组成，如图 9-1 所示。

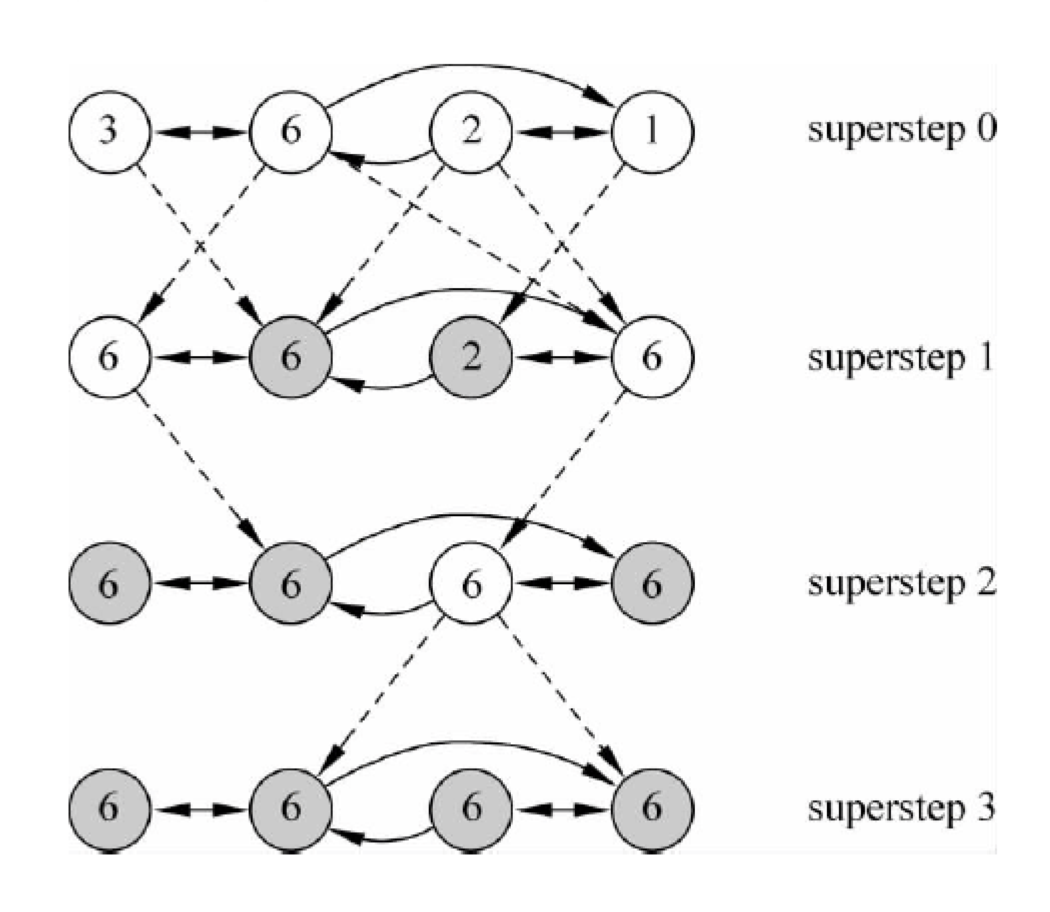


图 9-1　BSP 计算模型

从水平上看，在一个超步中所有的进程并行执行局部计算。一个超步可以分为 3 个阶段，如图 9-2 所示。

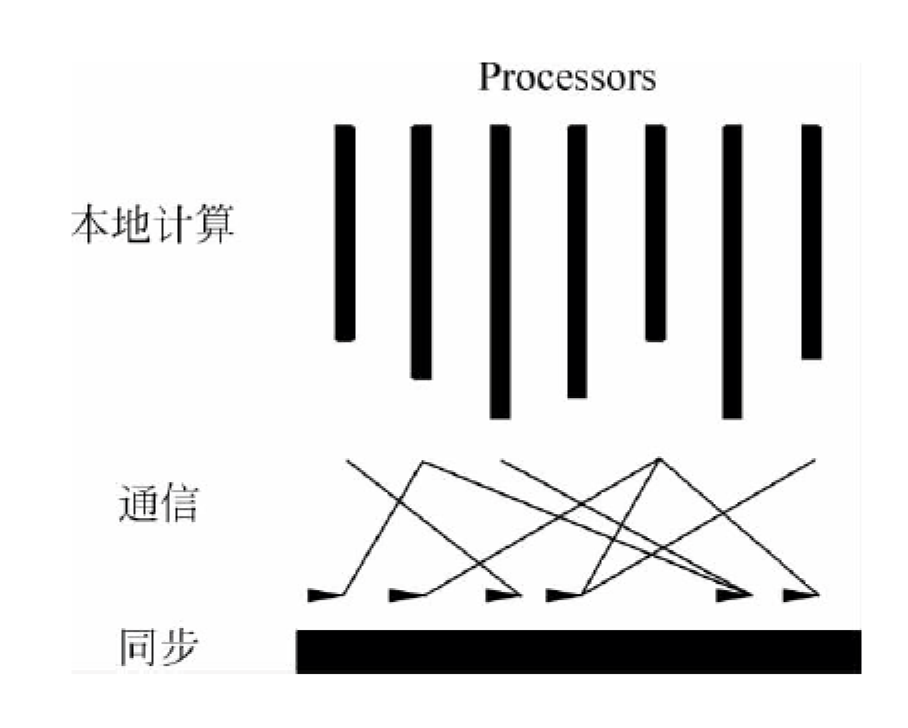


图 9-2　BSP 中的 superstep

（1）在本地计算阶段，每个处理器只对存储在本地内存中的数据进行本地计算。

（2）在全局通信阶段对任何非本地数据进行操作。

（3）在栅栏同步阶段等待所有通信行为的结束。

BSP 模型有以下几个特点。

（1）将计算划分为一个个的超步，有效地避免死锁。

（2）将处理器和路由器分开，强调了计算任务和通信任务的分离，而路由器仅完成点到点的消息传递，不提供组合、复制和广播等功能，这样做既掩盖了具体的互连网络拓扑，又简化了通信协议。

（3）采用障碍同步的方式，以硬件实现的全局同步是可控的粗粒度级，提供了执行紧耦合同步式并行算法的有效方式。

9.2　图计算与流计算、批处理的对比

图计算是以图论为基础、以图的方式抽象每个需要处理的计算问题，基本的数据结构表达就是：G＝（V，E，D），其中 V ＝ vertex（顶点或者节点），E＝edge（边），D＝data（权重）。例如，对于一个消费者的原始购买行为有用户和产品两类节点，边就是购买行为，权重是边上的一个数据结构，可以是购买次数和最后购买时间。对于我们面临的物理世界的许多数据问题都可以利用图结构来抽象表达，如社交网络、网页链接关系、用户传播网络，以及用户网络点击、浏览和购买行为，甚至消费者评论内容、内容分类标签、产品分类标签等。采用图抽象问题处理模型可以很好地表达现实生活中的各个事物之间的关联。另外，图计算任务之间的通信模型既包括同步模型（如 BSP 模型），也可以采用异步通信计算模型。

流计算处理一般是针对在线实时计算问题提出解决方法，重点强调问题处理的实时性，要求计算时延低。批处理方法一般是针对离线数据处理，强调的是批处理系统的吞吐量。从目前的离线处理框架 MapReduce、Spark 到实时计算框架 Hadoop Online、Spark Streaming 和 Flink，基本上采用的是 BSP 的同步模型。

并行图（graph-parallel）计算和实时计算、批处理计算类似，实时计算和批处理计算采用了一种 record-centric 的集合视图，而分布式图计算采用了一种 vertex-centric 的图视图。实时计算、批处理计算通过同时处理独立的数据来达到并发的目的，分布式图计算则是通过对图数据进行分区（即切分）来达到并发的目的。更准确地说，分布式图计算递归地定义特征的转换函数（这种转换函数作用于邻居特征），通过并发地执行这些转换函数来达到并发的目的。

总而言之，图计算、流计算（实时计算）、批处理计算都是针对现实中问题的解决思路，在实际问题中不断互补。

9.3 Spark GraphX

GraphX 是一个新的 Spark API，用于图（Graph）和并行图（graph-parallel）的计算。GraphX 通过引入弹性分布式属性图（Resilient Distributed Property Graph，顶点和边均有属性的有向多重图）来扩展 Spark RDD。为了支持图计算，GraphX 开发了一组基本的功能操作和一个优化过的 Pregel API。另外，GraphX 包含了一个快速增长的图算法和图 builders 的集合，用于简化图分析任务。

GraphX 的核心抽象是弹性分布式属性图，它是一个有向多重图，带有连接到每个顶点和边的用户定义的对象。在有向多重图中，多个并行的边共享相同的源和目的顶点。其支持并行边的能力简化了建模场景，相同的顶点可能存在多种关系（如 co-worker 和 friend）。另外每个顶点用一个唯一的 64 位长的标识符（VertexID）作为 key。GraphX 并没有对顶点标识强加任何排序。同样，边拥有相应的源和目的顶点标识符。

属性图扩展了 Spark RDD 的抽象，有 Table 和 Graph 两种视图，但是只需要一份物理存储。两种视图都有自己独有的操作符，从而使用户同时获得了操作的灵活性和执行的高效率。属性图以 vertex（VD）和 edge（ED）作为参数类型，这些类型分别是顶点和边相关联的对象的类型。

在某些情况下，在同样的图中，用户可能希望拥有不同属性类型的顶点，这可以通过继承完成。例如将用户和产品建模成一个二分图，可以使用以下方式：

```
class VertexProperty()
case class UserProperty(val name: String) extends VertexProperty
case class ProductProperty(val name: String, val price: Double) extends VertexProperty
//该图可能会有类型
var graph: Graph[VertexProperty, String] = null
```

和 RDD 一样，属性图是不可变的、分布式的、容错的。图的值或者结构的改变需要生成一个新的图来实现。注意，原始图中不受影响的部分都可以在新图中重用，用来减少存储的成本。执行者使用一系列顶点分区方法对图进行分区。和 RDD 一样，图的每个分区可以在发生故障的情况下被重新创建在不同的机器上。

在逻辑上，属性图对应于一对类型化的集合（RDD），这个集合包含每一个顶点和边的属性。因此，在图的类中包含访问图中顶点和边的成员变量。

```
class Graph[VD, ED] {
  val vertices: VertexRDD[VD]
  val edges: EdgeRDD[ED]
}
```

VertexRDD[VD]类和 EdgeRDD[ED]类是 RDD[(VertexID, VD)]和 RDD[Edge[ED]]的继承和优化版本。VertexRDD[VD]和 EdgeRDD[ED]都提供了额外的图计算功能及内部优化功能。

```
abstract class VertexRDD[VD](
    sc: SparkContext,
    deps: Seq[Dependency[_]]) extends RDD[(VertexId, VD)](sc, deps)

abstract class EdgeRDD[ED](
    sc: SparkContext,
    deps: Seq[Dependency[_]]) extends RDD[Edge[ED]](sc, deps)
```

在图存储模式方面,GraphX 借鉴 PowerGraph,使用 Vertex-Cut(点分割)方式存储图,用 3 个 RDD 存储图数据信息。

(1) VertexTable(id, data)。id 为顶点 id,data 为顶点属性。

(2) EdgeTable(pid, src, dst, data)。pid 为分区 id,src 为源顶点 id,dst 为目的顶点 id,data 为边属性。

(3) RoutingTable(id, pid)。id 为顶点 id,pid 为分区 id。

点分割存储的实现如图 9-3 所示。

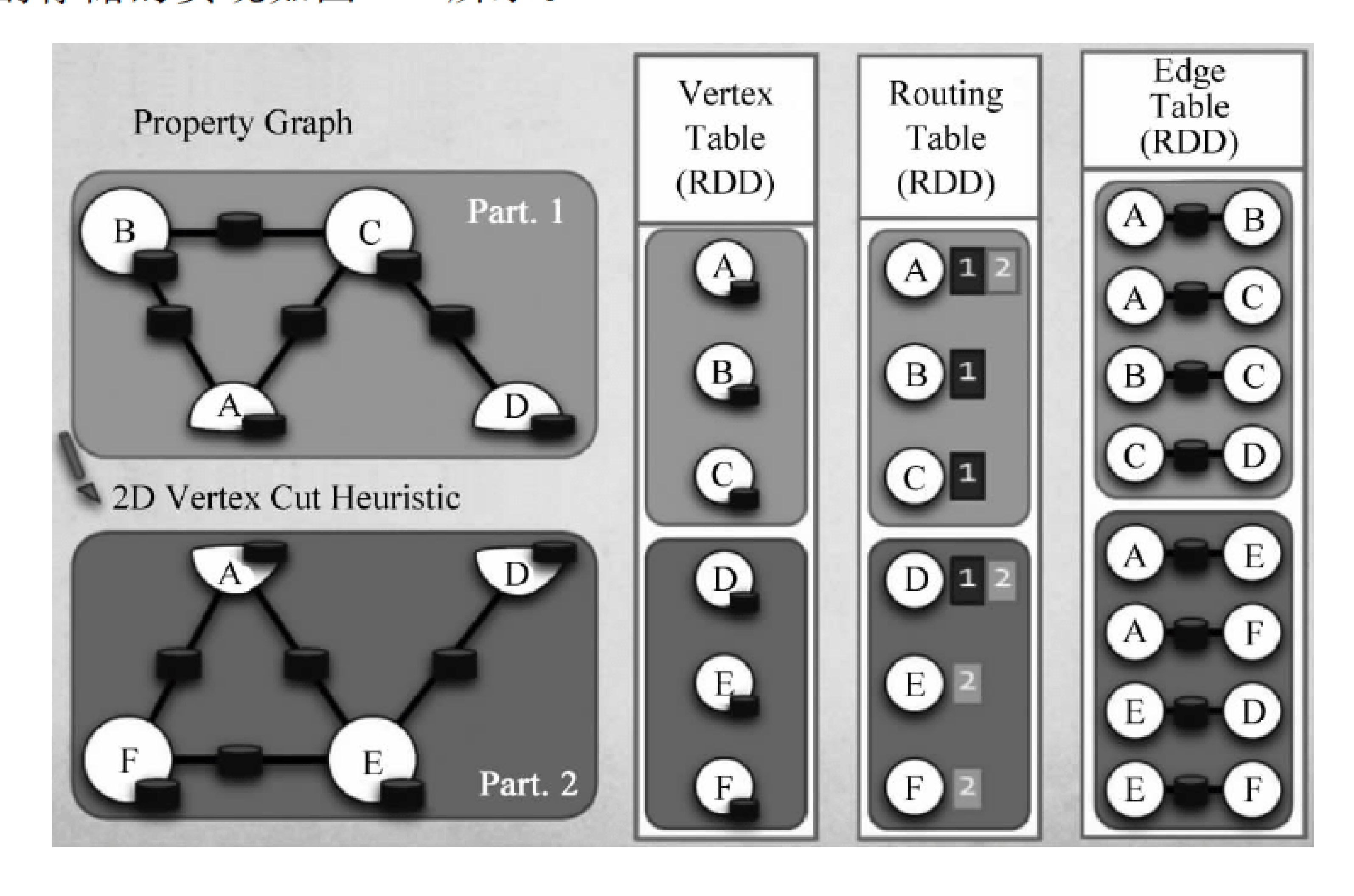


图 9-3 GraphX 点分割存储模型

9.4 Pregel

Pregel 是一种面向图算法的分布式编程框架，采用迭代的计算模型，即在每一轮，每个顶点处理上一轮收到的消息，并发出消息给其他顶点，更新自身状态和拓扑结构（出、入边）等。

在 Pregel 计算模式中，输入是一个有向图，该有向图的每一个顶点都有一个相应的、由字符串描述的 vertex identifier。每一个顶点都有一些属性，这些属性可以被修改，其初始值由用户定义。每一条有向边都和其源顶点关联，并且也拥有一些用户定义的属性和值，同时还记录了其目的顶点的 ID。

一个典型的 Pregel 计算过程如下：读取输入，初始化该图，当图被初始化好后运行一系列的超步，每一次超步都在全局的角度上独立运行，直到整个计算结束，输出结果。在每一次超步中，顶点的计算都是并行的，并且执行用户定义的同一个函数。每个顶点可以修改其自身的状态信息或以它为起点的出边的信息，从前序超步中接收消息，并传送给其后续超步，或者修改整个图的拓扑结构。边在这种计算模式中并不是核心对象，没有相应的计算运行在其上。

算法是否能够结束取决于是否所有的顶点都已经 vote 标识其自身达到 halt 状态了。在 superstep 0 中，所有顶点都置于 active 状态，每一个 active 的顶点都会在计算的执行中在某一次的超步中被计算。顶点通过将其自身的状态设置成 halt 来表示它已经不再 active。这就表示该顶点没有进一步的计算需要进行，除非被其他的运算触发，而 Pregel 框架将不会在接下来的超步中计算该顶点，除非该顶点收到一个其他超步传送的消息。如果顶点接收到消息，该消息将该顶点重新置为 active，那么在随后的计算中该顶点必须再次 deactive 其自身。整个计算在所有顶点都达到 inactive 状态并且没有消息在传送的时候宣告结束。这种简单的状态机制在图 9-4 中进行了描述。

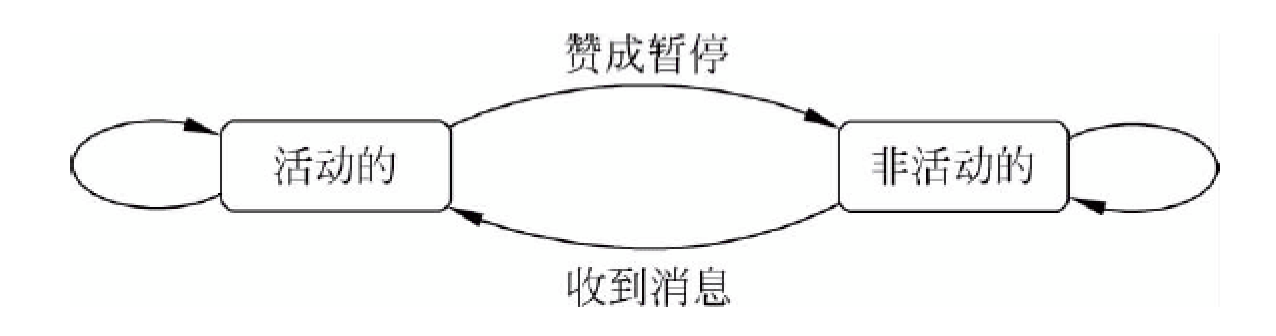


图 9-4 Pregel 的顶点状态变化

Pregel 选择了一种纯消息传递的模式，忽略远程数据读取和其他共享内存的方式，这样做有下面两个原因。

（1）消息的传递有足够高效的表达能力，不需要远程读取（remote reads）。

（2）对性能的考虑。在一个集群环境中，从远程机器上读取一个值会有很高的延迟，这种情况很难避免。消息传递模式通过异步和批量的方式传递消息，可以缓解这种远程读取的延迟。

图算法其实也可以写成一系列的链式 MapReduce 作业。选择不同模式的原因在于可用性和性能。Pregel 将顶点和边在本地机器上进行运算，仅利用网络来传输信息，而不是传输数据。MapReduce 本质上是面向函数的，所以将图算法用 MapReduce 来实现需要将整个图的状态从一个阶段传输到另一个阶段，这样就需要许多的通信和随之而来的序列化、反序列化的开销。另外，在一连串的 MapReduce 作业中各阶段需要协同工作也给编程增加了难度，这样的情况能够在 Pregel 的各轮超步的迭代中避免。

9.5　航班机场状态分析

图用来建立不同对象之间的模型。Spark GraphX 扩展了 Spark RDD 模型，采用了弹性分布式的属性图。一个属性图是一个有向多图，有多条同时并行的边，这样在多个相同的节点之间可以拥有多个关系。下面采用 GraphX 来分析航班机场状态，图 9-5 是一个简单的城市之间的航班信息图。

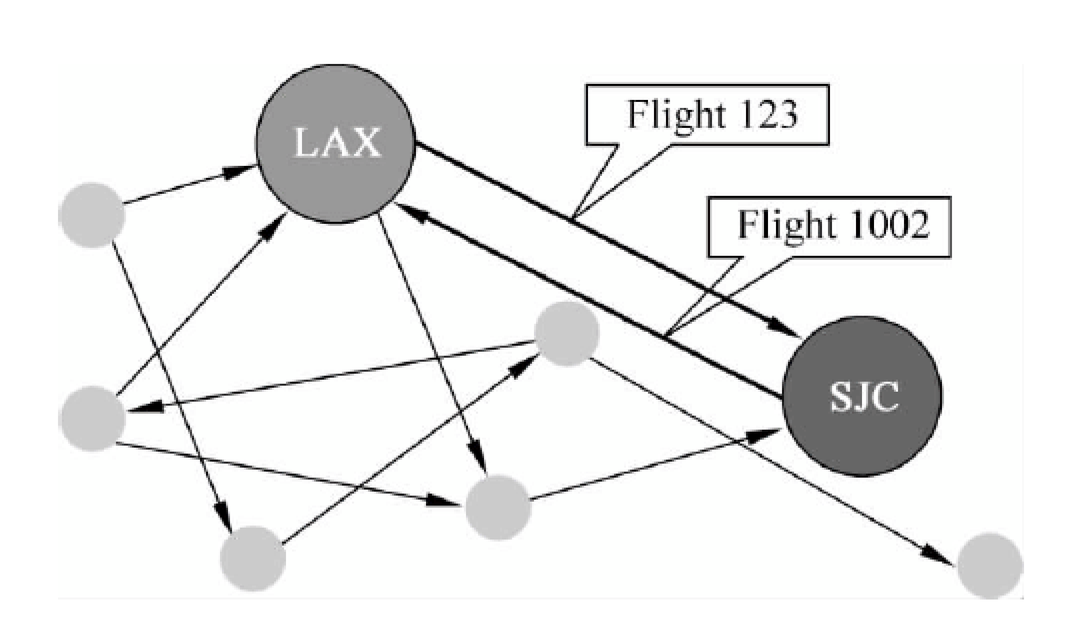


图 9-5　城市之间的航班信息

数据集来自 http://www.transtats.bts.gov/DL_SelectFields.asp?Table_ID=236&DB_Short_Name=On-Time，使用 2015 年 1 月份的航班数据，每个航班具有表 9-1 所示的信息。

表 9-1　航班样例信息

Field	Description	Example Value
dOfM(String)	Day of month	1
dOfW(String)	Day of week	4
carrier(String)	Carrier code	AA

续表

Field	Description	Example Value
tailNum(String)	Unique identifier for the plane-tail number	N787AA
flnum(Int)	Flight number	21
org_id(String)	Origin airport ID	12478
origin(String)	Origin airport code	JFK
dest_id(String)	Destination airport ID	12892
dest(String)	Destination airport code	LAX
crsdeptime(Double)	Scheduled departure time	900
deptime(Double)	Actual departure time	855
depdelaymins(Double)	Departure delay in minutes	0
crsarrtime(Double)	Scheduled arrival time	1230
arrtime(Double)	Actual arrival time	1237
arrdelaymins(Double)	Arrival delay minutes	7
crselapsedtime(Double)	Elapsed time	390
dist(Int)	Distance	2475

现在计算多少个机场有飞机飞出或者有飞机进入机场，并进行机场数据可视化分析。具体的源代码和数据在 GitHub 仓库中（https://github.com/alibook/alibook-bigdata.git）。

首先需要针对每个机场建立顶点表，部分机场顶点信息如表 9-2 所示。

接着构建路程表，部分机场路程信息如表 9-3 所示。

表 9-2　部分机场顶点信息

ID	Property
10397	ATL

表 9-3　部分机场路程信息

srcid	dstid	Property
14869	14683	1087

然后建立属性图，这样就可以分析有多少个航班、多少个机场、哪个机场最繁忙。具体的 spark-shell 命令见 GitHub 上的代码。

9.6　图计算的发展趋势

随着互联网技术的不断发展，社交网络分析、移动电话网络分析的规模变得越来越大，处理复杂度越来越高，也促进产生新的、更加高效的分布式图计算框架，以及更加高效的图计算算法。同时，以图作为问题模型的处理方式也在渐渐地扩展到其他领域。

9.7 习题

1. 简述图计算。

2. 什么是 BSP 计算模型？简述 BSP 的工作原理。

3. 简述 Pregel 的工作原理。

4. 采用 Spark GraphX 图计算框架编写一个单源点最短路径图计算程序，并在分布式环境下运行。

第 10 章 阿里云大数据计算服务平台

10.1　MaxCompute 概述

阿里云的大数据计算服务(MaxCompute,原名 ODPS)是一种快速的、完全托管的 TB/PB 级数据仓库解决方案。MaxCompute 向用户提供了完善的数据导入方案以及多种经典的分布式计算模型,能够更快速地解决用户海量数据计算问题,有效降低企业成本,并保障数据安全。

MaxCompute 主要服务于批量结构化数据的存储和计算,可以提供海量数据仓库的解决方案以及针对大数据的分析建模服务。随着社会数据收集手段的不断丰富及完善,越来越多的行业数据被积累下来。数据规模已经增长到了传统软件行业无法承载的海量数据(百 GB、TB 乃至 PB)级别。在分析海量数据场景下,由于单台服务器的处理能力的限制,数据分析者通常采用分布式计算模式。但分布式计算模型对数据分析人员提出了较高的要求,且不易维护。使用分布式计算模型,数据分析人员不仅需要了解业务需求,还需要熟悉底层计算模型。

MaxCompute 的开发目的是为用户提供一种便捷的分析处理海量数据的手段。用户可以不必关心分布式计算细节,而达到分析大数据的目的。

图 10-1 展示了 MaxCompute 平台的体系结构,数据通过 DataHub 接入到计算引擎。不同于其他的计算分布式系统,MaxCompute 还分割管理层和运算层,管理层封装底层的多个计算集群,使得计算引擎可以当作一个运算平台,可以打破地域的限制,做到真正的跨地域、跨机房的大型运算平台。还有一个重要的原因是基于安全性的要求。系统只

在计算集群内执行用户自定义的函数，而在管理层进行用户权限的检查，在利用沙箱技术隔离恶意用户代码的同时，通过网段隔离进一步保障用户数据的安全性。

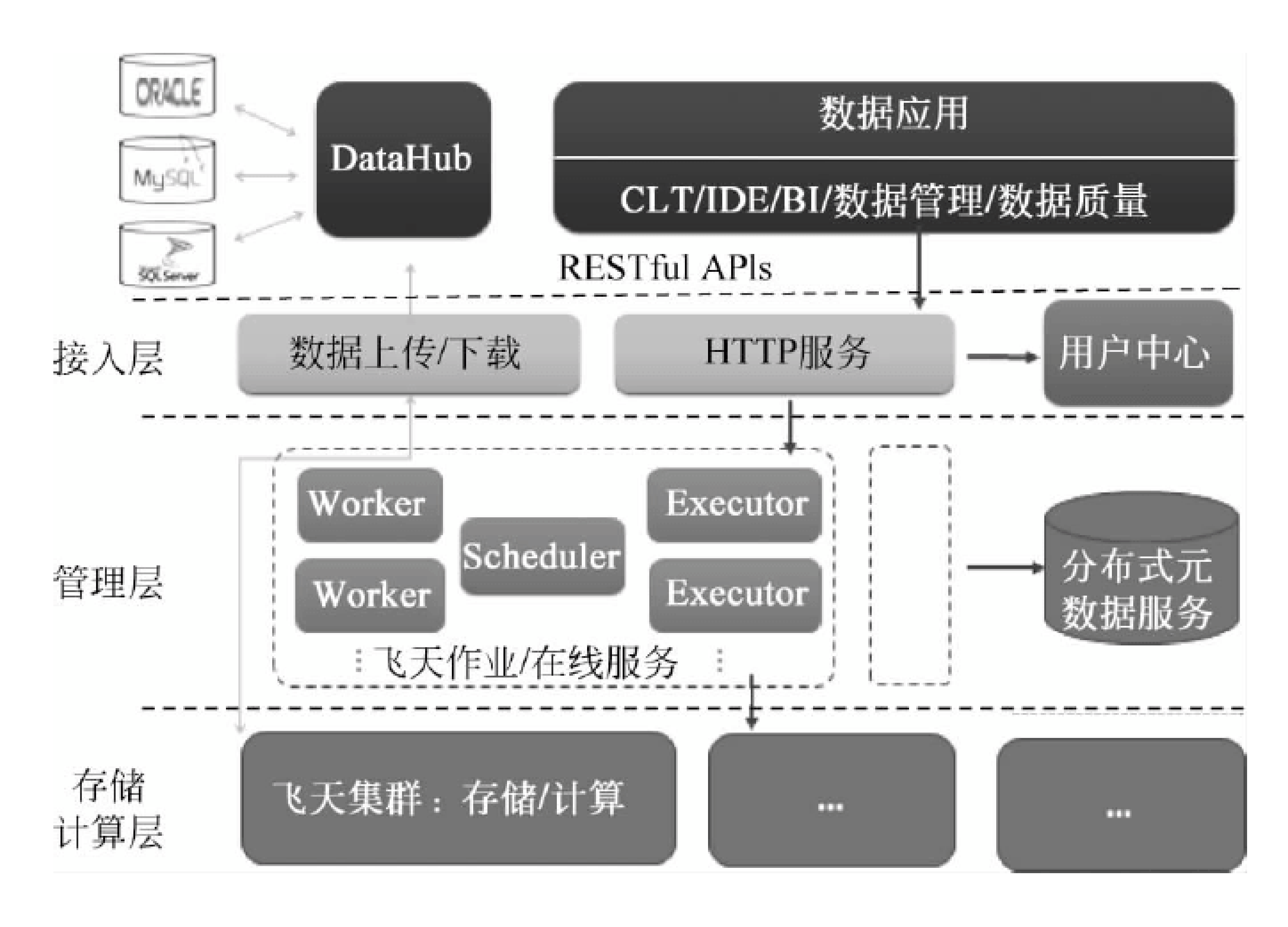


图 10-1　MaxCompute 系统架构

10.2　MR 计算

阿里云的 MR 计算是 MaxCompute 提供的 Java MapReduce 编程模型。值得注意的是，由于 MaxCompute 并没有开放文件接口，用户只能通过它所提供的 Table 读写数据，因此 MaxCompute 的 MapReduce 模型与开源社区中通用的 MapReduce 模型在使用上有一定的区别。这样的改动虽然失去了一定的灵活性，例如不能够自定义排序及哈希算法，但却能够简化开发流程，免除很多琐碎的工作。更为重要的是，MaxCompute 还提供了基于 MapReduce 的扩展计算模型，即 MR2。在该模型下，一个 Map 函数后可以接入连续的多个 Reduce 函数。

HDFS 是一个分布式文件系统，提供了可靠存储功能，运行在廉价服务器上。HDFS 集群中 Client 和 NameNode 之间的大部分通信通过 RPC 操作来完成，DataNode 和 NameNode 之间的心跳机制也是通过 RPC 操作来完成，因此 RPC 是 HDFS 运行组件的重要部分。这里需要分析 HDFS 集群在运行过程中产生了多少次 RPC 操作。

思路很简单，可以分析 HDFS 集群的 NameNode 运行日志。开启 HDFS 集群 NameNode 的日志，log 级别为 debug 级别，这样可以看到 RPC 操作的具体信息。因为每次 RPC 操作通过 Hadoop IPC 工具操作，分析时会产生包含 processOneRpc 这样的关键字的一条日志记录，所以，找出 NameNode 日志文件中包含 processOneRpc 的记录数

目就是产生的 RPC 操作数目。利用 MapReduce，在 Map 阶段针对读取的每个记录进行关键字匹配，如果匹配，就生成关键字对< processOneRpc，1 >，然后在 Reduce 阶段对相同关键字求和。为了容易读取，对产生的关键字对逆转。该程序提供了类似 Linux 下 grep 命令的功能，可以查找匹配的字符串。

对应的代码如下，详细的代码可以在 GitHub（https://github.com/alibook/alibook-bigdata.git）上找到，下载后按照下面的操作方式就可以运行。

```
public class odpsgrep {

  /**
   * RegexMapper
   **/
  public static class RegexMapper extends MapperBase {
    private Pattern pattern;
    private int group;

    private Record word;
    private Record one;


    @Override
    public void setup(TaskContext context) throws IOException {
      JobConf job = (JobConf) context.getJobConf();
      pattern = Pattern.compile(job.get("mapred.mapper.regex"));
      group = job.getInt("mapred.mapper.regex.group", 0);

      word = context.createMapOutputKeyRecord();
      one = context.createMapOutputValueRecord();
      one.set(new Object[] { 1L });
    }

    @Override
    public void map(long recordNum, Record record, TaskContext context) throws IOException {
      for (int i = 0; i < record.getColumnCount(); ++i) {
        String text = record.get(i).toString();
        Matcher matcher = pattern.matcher(text);
        while (matcher.find()) {
```

```
                word.set(new Object[] { matcher.group(group) });
                context.write(word, one);
            }
        }
    }
}

/**
 * LongSumReducer
 **/
public static class LongSumReducer extends ReducerBase {
    private Record result = null;

    @Override
    public void setup(TaskContext context) throws IOException {
        result = context.createOutputRecord();
    }

    @Override
    public void reduce(Record key, Iterator<Record> values, TaskContext context) throws
IOException {
        long count = 0;
        while (values.hasNext()) {
            Record val = values.next();
            count += (Long) val.get(0);
        }
        result.set(0, key.get(0));
        result.set(1, count);
        context.write(result);
    }
}

/**
 * 一个{@link Mapper},交换键和值
 **/
public static class InverseMapper extends MapperBase {
    private Record word;
    private Record count;
```

```
    @Override
    public void setup(TaskContext context) throws IOException {
      word = context.createMapOutputValueRecord();
      count = context.createMapOutputKeyRecord();
    }

    /**
     * 翻转函数,将输入的键和值交换
     **/
    @Override
    public void map(long recordNum, Record record, TaskContext context) throws IOException
{
      word.set(new Object[] { record.get(0).toString() });
      count.set(new Object[] { (Long) record.get(1) });
      context.write(count, word);
    }
  }

  /**
   * IdentityReducer
   **/
  public static class IdentityReducer extends ReducerBase {
    private Record result = null;

    @Override
    public void setup(TaskContext context) throws IOException {
      result = context.createOutputRecord();
    }

    /** 将所有键和值写到输出 **/

    @Override
    public void reduce(Record key, Iterator<Record> values, TaskContext context) throws
IOException {
      result.set(0, key.get(0));

      while (values.hasNext()) {
        Record val = values.next();
        result.set(1, val.get(0));
```

```
            context.write(result);
          }
        }
      }

      public static void main(String[] args) throws Exception {
        if (args.length < 4) {
          System.err.println("Grep <inDir><tmpDir><outDir><regex> [<group>]");
          System.exit(2);
        }

        JobConf grepJob = new JobConf();

        grepJob.setMapperClass(RegexMapper.class);
        grepJob.setReducerClass(LongSumReducer.class);

        grepJob.setMapOutputKeySchema(SchemaUtils.
            fromString("word:string"));
        grepJob.setMapOutputValueSchema(SchemaUtils.
            fromString("count:bigint"));

        InputUtils.addTable(TableInfo.builder().
            tableName(args[0]).build(), grepJob);
        OutputUtils.addTable(TableInfo.builder().
            tableName(args[1]).build(), grepJob);

        grepJob.set("mapred.mapper.regex", args[3]);
        if (args.length == 5) {
          grepJob.set("mapred.mapper.regex.group", args[4]);
        }

        RunningJob rjGrep = JobClient.runJob(grepJob);

        JobConf sortJob = new JobConf();

        sortJob.setMapperClass(InverseMapper.class);
        sortJob.setReducerClass(IdentityReducer.class);

        sortJob.setMapOutputKeySchema(SchemaUtils.
```

```
            fromString("count:bigint"));
        sortJob.setMapOutputValueSchema(SchemaUtils.
            fromString("word:string"));

        InputUtils.addTable(TableInfo.builder().
          tableName(args[1]).build(), sortJob);
        OutputUtils.addTable(TableInfo.builder().
          tableName(args[2]).build(), sortJob);

        sortJob.setNumReduceTasks(1);     //写一个文件
        sortJob.setOutputKeySortColumns(new String[] { "count" });

        RunningJob rjSort = JobClient.runJob(sortJob);
    }

}
```

上述示例代码的输入数据是 HDFS NameNode 节点运行日志文件,需要使用 awk 转换一下。输出结果也就是 HDFS 集群的 RPC 操作次数,存放在 alibook_grep_out 表中。

然后编译好 JAR 文件,生成 odpsgrep-0.0.1.jar 测试包文件。接下来是准备好统计 RPC 操作次数的测试表和环境。(注意,下面使用的 ODPS 是 MaxCompute 的原来的文字)

1. 下载 ODPS 客户端

在阿里云官网下载 odps_public 客户端(http://repo.aliyun.com/download/odpscmd/0.24.1/odpscmd_public.zip?spm=5176.doc27991.2.2.6bt8vL&file=odpscmd_public.zip),在文件 conf/odps_config.ini 中更改 Access Key 和 Access ID 信息,添加 project 名字(需要在 MaxCompute 上创建一个 project)。在本实验中建立的 project 名字为 aliyunbook。

图 10-2 所示为 Access Key 信息的页面。

图 10-2　Access Key 信息的页面

2. 创建表

在该统计 RPC 操作次数的测试用例中，输入数据和输出结果都存放在表中，因此需要创建表。启动 ODPS 客户端创建表，如图 10-3 和图 10-4 所示。

```
[qzhong@dell122 bin]$ ./odpscmd
Aliyun ODPS Command Line Tool
Version 0.24.1
@Copyright 2015 Alibaba Cloud Computing Co., Ltd. All rights reserved.
odps@ aliyunbook>
```

图 10-3 启动 ODPS 客户端

```
odps@ aliyunbook>create table alibook_grep_src(line string);
OK
odps@ aliyunbook>create table alibook_grep_tmp(key string, cnt bigint);
OK
odps@ aliyunbook>create table alibook_grep_out(key bigint, value string);
OK
```

图 10-4 创建输入和输出表文件

3. 添加资源

需要上传本地编译好的 JAR 文件，如图 10-5 所示。

```
odps@ aliyunbook>add jar ~/odpsgrep-0.0.1.jar -f;
OK: Resource 'odpsgrep-0.0.1.jar' have been updated.
```

图 10-5 上传 JAR 文件

接着使用 awk 转换 NameNode 运行日志文件的数据格式，如图 10-6 所示。

```
awk '{printf("%s %s\n", $5, $8)}' ~/hadoop-namenode.log > tmp1.log
```

图 10-6 使用 awk 转换日志数据格式

然后使用 tunnel 导入数据，如图 10-7 所示。

```
odps@ aliyunbook>tunnel upload tmp1.log alibook_grep_src;
Upload session: 20161126223120c6399a0a01358f53
Start upload:tmp1.log
Using \n to split records
Total bytes:4523198      Split input to 1 blocks
2016-11-26 22:31:15     scan block: '1'
2016-11-26 22:31:15     scan block complete, blockid=1
2016-11-26 22:31:15     upload block: '1'
2016-11-26 22:31:16     upload block complete, blockid=1
upload complete, average speed is 631 KB/s
OK
```

图 10-7 导入数据到 alibook_grep_src 表中

4. 运行 WordCount

在 ODPS 中执行 grep 操作，如图 10-8 所示。

```
odps@ aliyunbook>jar -resources odpsgrep-0.0.1.jar -classpath /home/qzhong/odpsgrep-0.0.1.jar alibook.odpsgrep.o
dpsgrep alibook_grep_src alibook_grep_tmp alibook_grep_out processOneRpc;
```

图 10-8 执行 grep 操作

运行过程和结果如图 10-9 和图 10-10 所示。

```
2016-11-26 22:37:43 M1_job0:0/0/1[0%]   R2_1_job0:0/0/1[0%]
2016-11-26 22:37:51 M1_job0:0/1/1[100%] R2_1_job0:0/0/1[0%]
2016-11-26 22:38:00 M1_job0:0/1/1[100%] R2_1_job0:0/1/1[100%]
...
Inputs:
        aliyunbook.alibook_grep_src: 99271 (107416 bytes)
Outputs:
        aliyunbook.alibook_grep_tmp: 1 (560 bytes)
M1_aliyunbook_20161126143732930gbkur8jc2_LOT_0_0_0_job0:
        Worker Count:1
        Input Records:
                input: 99271 (min: 99271, max: 99271, avg: 99271)
        Output Records:
                R2_1: 15580 (min: 15580, max: 15580, avg: 15580)
R2_1_aliyunbook_20161126143732930gbkur8jc2_LOT_0_0_0_job0:
        Worker Count:1
        Input Records:
                input: 15580 (min: 15580, max: 15580, avg: 15580)
        Output Records:
```

图 10-9 grep 运行过程

```
odps@ aliyunbook>read alibook_grep_out;
+------------+------------+
| key        | value      |
+------------+------------+
| 15580      | processOneRpc |
+------------+------------+
odps@ aliyunbook>
```

图 10-10 grep 运行结果

如图 10-10 所示，在当前的 HDFS 集群中运行了 15 580 次 RPC 操作。

10.3 SQL 计算

MaxCompute SQL 采用标准的 SQL 语法、更高效的计算框架支持 SQL 计算模型，执行效率比普通的 MapReduce 模型更高。需要注意的是，MaxCompute SQL 不支持事

务、索引及Update/Delete等操作。

下面简单介绍一个SQL语句操作实例。

1. 创建表的操作

创建表sale_detail保存销售记录，该表使用销售时间(sale_date)和销售区域(region)作为分区列。

```
create table if not exists sale_detail(
shop_name string,
customer_id string,
total_price double)
partitioned by (sale_date string,region string);
-- 创建一张分区表 sale_detail
```

2. 删除表的操作

其代码如下：

```
create table sale_detail_drop like sale_detail;
drop table sale_detail_drop;
--若表存在，成功返回；若不存在，异常返回
drop table if exists sale_detail_drop2;
--无论是否存在 sale_detail_drop2 表，均成功返回
```

3. 更新表中的数据

其代码如下：

```
insert overwrite table sale_detail_insert partition (sale_date = '2013', region = 'china')
select customer_id, shop_name, total_price from sale_detail;
-- 在创建 sale_detail_insert 表时，列的顺序为 shop_name string—customer_id string—
total_price bigint
-- 而从 sale_detail 向 sale_detail_insert 插入数据时，sale_detail 的插入顺序为 customer_
id—shop_name—total_price
-- 此时会将 sale_detail.customer_id 的数据插入 sale_detail_insert.shop_name
-- 将 sale_detail.shop_name 的数据插入 sale_detail_insert.customer_id
```

10.4 Graph 计算

对于某些复杂的迭代计算场景，例如 K-Means、PageRank 等，如果仍然使用 MapReduce 来完成这些计算任务将是非常耗时的。MaxCompute 提供的 Graph 模型能够非常好地完成这一类计算任务。

互联网的快速发展给人们提供了很大的便利，人们可以使用互联网提供的便捷快速订购到自己喜欢的东西，外卖业务的发展使人们可以足不出户订购到自己喜欢的美食。对于外卖配送人员而言，以最快的速度把美食送到客户手中可以赚更多的钱；对于商家而言，将订单物品快速送到客户手中可以吸引更多的客户。

下面以外卖配送人员为例，求出他把订单物品送到客户手中的最短路径。采用 dijstra 方法计算，求取单源点最短路径。dijstra 方法是求解单源点最短路径的经典算法。

最短距离：对于一个有权重的有向图 G=(V,E)，从一个源点 s 到汇点 v 有很多条路径，其中边权之和最小的路径称作从 s 到 v 的最短距离。

算法基本原理如下。

(1) 初始化。源点 s 到 s 自身的距离(d[s]=0)，其他点 u 到 s 的距离为无穷(d[u]=∞)。

(2) 迭代。若存在一条从 u 到 v 的边，那么从 s 到 v 的最短距离更新为 d[v]=min(d[v], d[u]+weight(u, v))，直到所有的点到 s 的距离不再发生变化时迭代结束。

使用 MaxCompute Graph 程序进行求解：每个点维护其到源点的当前最短距离值，当这个值变化时，将新值加上边的权值发送消息通知其邻接点，在下一轮迭代时，邻接点根据收到的消息更新其当前最短距离，当所有点的当前最短距离不再变化时迭代终止。

部分代码如下：

```
public class graph {
  public static final String START_VERTEX = "sssp.start.vertex.id";
  public static class SSSPVertex extends
      Vertex < LongWritable, LongWritable, LongWritable, LongWritable > {
    private static long startVertexId = -1;
    public SSSPVertex() {
      this.setValue(new LongWritable(Long.MAX_VALUE));
    }
    public boolean isStartVertex(
        ComputeContext < LongWritable, LongWritable, LongWritable, LongWritable > context) {
```

```
      if (startVertexId == -1) {
        String s = context.getConfiguration().get(START_VERTEX);
        startVertexId = Long.parseLong(s);
      }
      return getId().get() == startVertexId;
    }
    @Override
    public void compute(
        ComputeContext<LongWritable, LongWritable, LongWritable, LongWritable> context,
        Iterable<LongWritable> messages) throws IOException {
long minDist = isStartVertex(context) ? 0 : Integer.MAX_VALUE;
      for (LongWritable msg : messages) {
        if (msg.get() < minDist) {
          minDist = msg.get();
        }
      }
      if (minDist < this.getValue().get()) {
        this.setValue(new LongWritable(minDist));
        if (hasEdges()) {
          for (Edge<LongWritable, LongWritable> e : this.getEdges()) {
            context.sendMessage(e.getDestVertexId(), new LongWritable(minDist
                + e.getValue().get()));
          }
        }
      } else {
        voteToHalt();
      }
    }
    @Override
    public void cleanup(
        WorkerContext<LongWritable, LongWritable, LongWritable, LongWritable> context)
        throws IOException {
      context.write(getId(), getValue());
    }
  }
  public static class MinLongCombiner extends
      Combiner<LongWritable, LongWritable> {
    @Override
    public void combine(LongWritable vertexId, LongWritable combinedMessage,
        LongWritable messageToCombine) throws IOException {
      if (combinedMessage.get() > messageToCombine.get()) {
        combinedMessage.set(messageToCombine.get());
```

```
          }
        }
      }
      public static class SSSPVertexReader extends
          GraphLoader<LongWritable, LongWritable, LongWritable, LongWritable> {
        @Override
        public void load(
            LongWritable recordNum,
            WritableRecord record,
            MutationContext<LongWritable, LongWritable, LongWritable, LongWritable> context)
            throws IOException {
          SSSPVertex vertex = new SSSPVertex();
          vertex.setId((LongWritable) record.get(0));
          String[] edges = record.get(1).toString().split(",");
          for (int i = 0; i < edges.length; i++) {
            String[] ss = edges[i].split(":");
            vertex.addEdge(new LongWritable(Long.parseLong(ss[0])),
                new LongWritable(Long.parseLong(ss[1])));
          }
          context.addVertexRequest(vertex);
        }
      }
      public static void main(String[] args) throws IOException {
        if (args.length < 2) {
          System.out.println("Usage: <startnode><input><output>");
          System.exit(-1);
        }
        GraphJob job = new GraphJob();
        job.setGraphLoaderClass(SSSPVertexReader.class);
        job.setVertexClass(SSSPVertex.class);
        job.setCombinerClass(MinLongCombiner.class);
        job.set(START_VERTEX, args[0]);
        job.addInput(TableInfo.builder().tableName(args[1]).build());
        job.addOutput(TableInfo.builder().tableName(args[2]).build());
        long startTime = System.currentTimeMillis();
        job.run();
        System.out.println("Job Finished in "
            + (System.currentTimeMillis() - startTime) / 1000.0 + " seconds");
      }
    }
```

以上是采用 MaxCompute Graph 计算单源点最短路径的代码，详细的可运行代码可以从 GitHub 上下载(https://github.com/alibook/alibook-bigdata.git)。

运行以上代码，首先要创建表文件，如图 10-11 所示。

```
odps@ aliyunbook>create table alibook_sssp_in(v bigint, es string);
OK
odps@ aliyunbook>create table alibook_sssp_out(v bigint, l bigint);
OK
```

图 10-11　创建数据表文件

然后上传数据文件，如图 10-12 所示。

```
odps@ aliyunbook>tunnel u -fd " " alibook_sssp.txt alibook_sssp_in;
Upload session: 20161126231815293c9b0a00af5257
Start upload:alibook_sssp.txt
Using \n to split records
Total bytes:66   Split input to 1 blocks
2016-11-26 23:18:10     scan block: '1'
2016-11-26 23:18:10     scan block complete, blockid=1
2016-11-26 23:18:10     upload block: '1'
2016-11-26 23:18:11     upload block complete, blockid=1
upload complete, average speed is 66 bytes/s
OK
```

图 10-12　上传数据文件

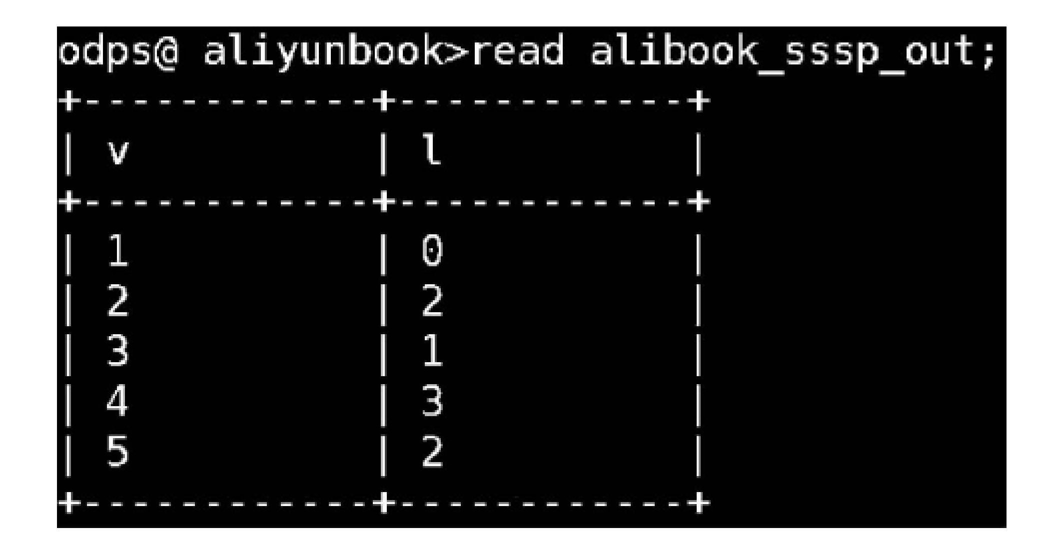
启动 ODPS(即 MaxCompute)任务运行分析，如图 10-13 所示。

```
odps@ aliyunbook>jar -libjars graph-0.0.1.jar -classpath /home/qzhong/graph-0.0.1.jar alibook.graph.graph 1 alib
ook_sssp_in alibook_sssp_out;
Running job in console.

ID = 20161126153428641geial2i8
http://logview.odps.aliyun.com/logview/?h=http://service.odps.aliyun.com/api&p=aliyunbook&i=20161126153428641gei
al2i8&token=TzR5emJMQTV3R2JvUTlQWERhTzQyaTBOdHNZPSxPRFBTX09CTzoxNjAyNDE3ODg0NzIzNTU5LDE0ODA3NzkyNjgseyJTdGF0ZW1l
bnQiOlt7IkFjdGlvbiI6WyJvZHBzOlJlYWQiXSwiRWZmZWN0IjoiQWxsb3ciLCJSZXNvdXJjZSI6WyJhY3M6b2RwczoqOnByb2plY3RzL2FsaXll
bmJvb2svaW5zdGFuY2VzLzIwMTYxMTI2MTUzNDI4NjQxZ2VpYWwyaTgiXX1dLCJWZXJzaW9uIjoiMSJ9
2016-11-26 23:34:24 RUNNING     WAIT_WORKER_UP
2016-11-26 23:34:38 TERMINATED  TERMINATE
```

图 10-13　运行图计算任务

图 10-14 是最短路径计算结果，经计算配送员 1 到客户 5 的最短路径是 2km。



```
odps@ aliyunbook>read alibook_sssp_out;
+------------+------------+
| v          | l          |
+------------+------------+
| 1          | 0          |
| 2          | 2          |
| 3          | 1          |
| 4          | 3          |
| 5          | 2          |
+------------+------------+
```

图 10-14　最短路径计算结果

10.5 习题

1. 简述 MaxCompute 的功能和特性。

2. 利用 MaxCompute 编写 MR 程序，分析分布式系统运行日志，统计出现 warning 信息和 error 信息的次数。

3. 利用 MaxCompute 编写 Graph 计算程序，计算从一个城市到另一个城市的最短航班距离。

第 11 章

集群资源管理与调度

随着互联网的快速发展和大数据的来临，基于数据密集型应用的集群计算框架不断涌现，并且这些计算框架都只面向某一类特定领域的应用。基于这一特点，互联网公司往往需要部署和运行多个计算框架，从而为每个应用选择最优的计算框架。因此，资源统一管理和调度系统作为集群共享平台被提出来。当前比较有名的开源资源统一管理和调度平台有两个，一个是 Mesos，另外一个是 YARN。集群资源统一管理和调度系统需要同时支持多种不同的计算框架，如何管理集群计算资源和不同计算框架间的资源公平分配成为关键技术难点。不同计算框架的作业是异构的，如何在不同框架间进行作业调度以充分利用集群资源和提高系统吞吐量成为了新的挑战。

相比于“一种计算框架一个集群”的模式，共享集群的模式具有以下 3 个优点：

(1) 硬件共享，资源利用率高。如果每个框架一个集群，则往往由于应用程序的数量和资源需求的不均衡使得在某段时间内有些计算框架的资源紧张，而另外一些集群资源比较空闲。共享集群模式则通过多框架共享资源，使得集群中的资源得到更加充分的利用。

(2) 人员共享，运维成本低。采用“一种计算框架一个集群”的模式可能需要多个管理员来管理和维护集群，进而增加运维成本，而在共享模式下只需要少数几个管理员即可完成多个框架的统一管理。

(3) 数据共享，数据复制开销低。随着数据量的暴增，跨集群的数据移动不仅需要花费更长的时间，且硬件成本也会随之增加；而共享集群可让多个框架共享数据和硬件存储资源，这将大大减少数据复制的开销。

11.1 集群资源统一管理系统

简而言之，集群资源统一管理系统需要支持多种计算框架，并需要具有扩展性、容错性和高资源利用率等几个特点。一个行之有效的资源统一管理系统需要包含资源管理、分配和调度等功能。图11-1是统一管理与调度系统的基本架构图。

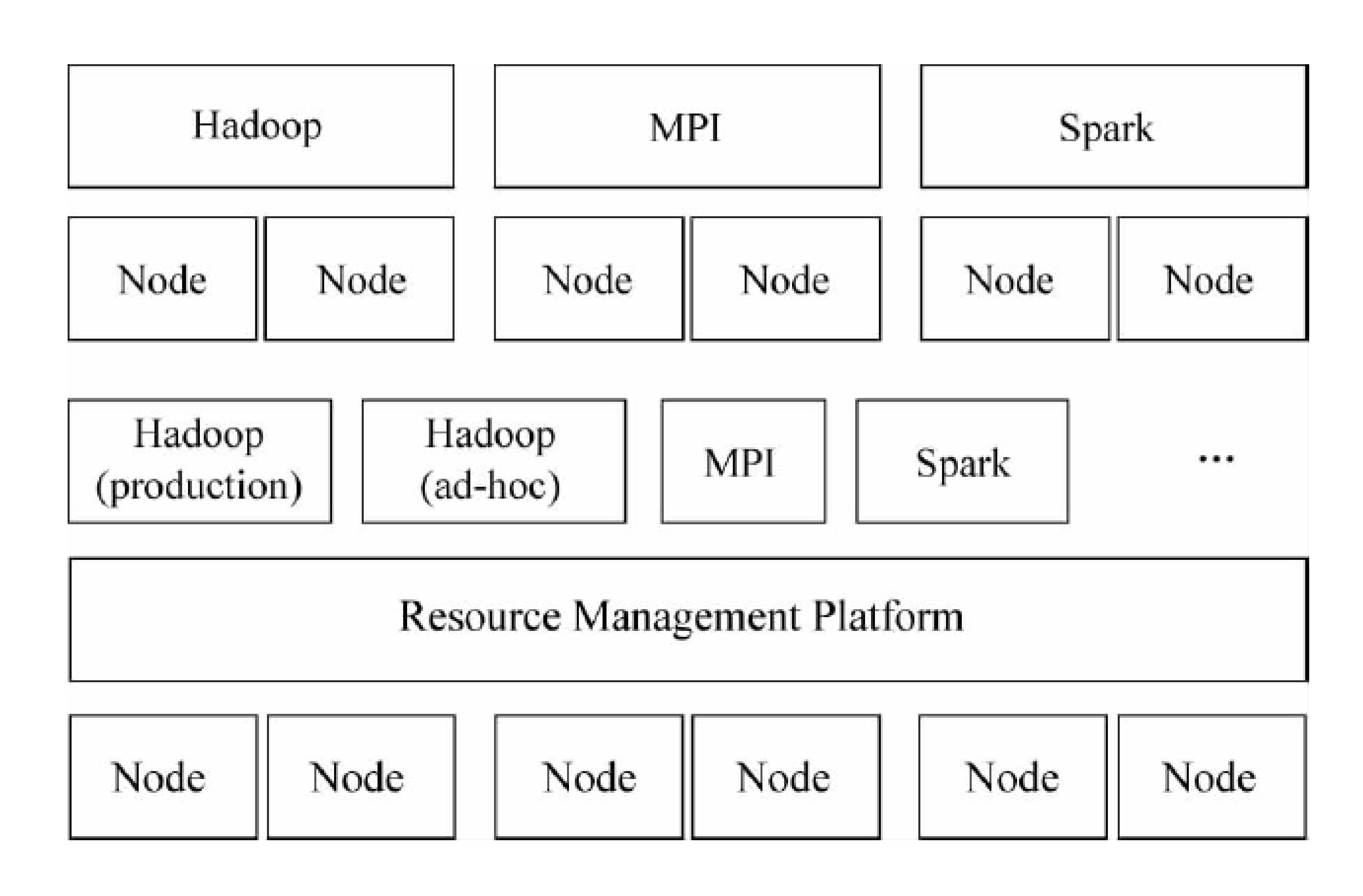


图11-1 资源统一管理与调度系统基本架构

基于真实资源需求的资源管理方案能够提升集群资源的利用率，进而提升吞吐量。

11.1.1 集群资源管理概述

商业服务器集群目前已经成为主要的计算平台，为互联网服务和大量的数据密集型科学计算提供了强大的计算能力。基于上述需求，研究人员和开发人员设计和实现了大量的分布式计算框架，简化集群程序的编写，最典型的例子包括MapReduce、Dryad、MapReduce Online（支持流任务）、Pregel（图计算框架）。新的计算框架仍然不断地产生，但是没有一种计算框架可以适合所有的计算任务，所以目前采取的方式是在同一个集群上运行多个计算框架，选取一个最优的。多个计算框架之间共享一个服务器集群可以共享大规模数据集，极大地降低因为数据集规模巨大而带来的复制开销。

当前多个计算框架共用一个服务器集群的方式是对集群进行静态划分，每个分区运行一个计算框架；另外一种方式是为每个计算框架分配一些虚拟机VM，但是这些方法都没有实现高利用率和数据共享，最重要的原因是当前这些解决方法的资源分配粒度和当前的计算框架不匹配。典型的计算框架，如Hadoop和Dryad，采用的是细粒度的资源共享模型，计算节点把资源划分成多个“slot”，并且一个Job由多个短任务task组成，短

任务实现了资源的高利用率和高扩展性，但是目前的大多数计算框架基本上都是独立开发，没有一个在多个计算框架之间细粒度共享资源的方式，从而在这些计算框架之间共享资源和数据变得更加困难和复杂。

因此设计一种集群资源管理系统支持多个计算框架，实现集群资源共享和高利用率。为了实现这一目标需要解决以下问题：

（1）支持多种不同的计算框架。不同的计算框架采用的是不同的资源共享模型、不同的资源调度需求、不同的通信模式以及不同的任务依赖关系，既要支持当前计算框架，也需要支持以后的计算框架。

（2）集群资源管理系统需要支持良好的扩展性。当前的集群资源拥有几万台计算节点，运行着几百个 Job，一次有几百万个 task 同时运行。

（3）需要具有良好的容错和高可靠性。

11.1.2 Apache YARN

Apache Hadoop YARN（Yet Another Resource Negotiator，另一种资源协调者）是一种新的 Hadoop 资源管理器，它是一个通用资源管理系统，可为上层应用提供统一的资源管理和调度，它的引入为集群在利用率、资源统一管理和数据共享等方面带来了巨大的好处。

YARN 的基本思想是将 JobTracker 的两个主要功能（资源管理和作业调度/监控）分离，主要方法是创建一个全局的 ResourceManager（RM）和若干个针对应用程序的 ApplicationMaster（AM）。这里的应用程序是指传统的 MapReduce 作业或 DAG（有向无环图）作业。

YARN 分层结构的本质是 ResourceManager。这个实体控制整个集群并管理应用程序向基础计算资源的分配。ResourceManager 将各个资源部分（计算、内存、带宽等）精心地安排给基础 NodeManager（YARN 的每个节点代理）。ResourceManager 还与 ApplicationMaster 一起分配资源，与 NodeManager 一起启动和监视它们的基础应用程序。在此上下文中，ApplicationMaster 承担了以前的 TaskTracker 的一些角色，ResourceManager 承担了 JobTracker 的角色。

ApplicationMaster 管理一个在 YARN 内运行的应用程序的每个实例。ApplicationMaster 负责协调来自 ResourceManager 的资源，并通过 NodeManager 监视容器的执行和资源使用（CPU、内存等的资源分配）。

NodeManager 管理一个 YARN 集群中的每个节点。NodeManager 提供针对集群中每个节点的服务，从监督对一个容器的终生管理到监视资源和跟踪节点健康。MRv1 通

过插槽管理 Map 和 Reduce 任务的执行，而 NodeManager 管理抽象容器，这些容器代表着可供一个特定应用程序使用的针对每个节点的资源。如果要使用一个 YARN 集群，首先需要来自包含一个应用程序的客户的请求。ResourceManager 协商一个容器的必要资源，启动一个 ApplicationMaster 来表示已提交的应用程序。通过使用一个资源请求协议，ApplicationMaster 协商每个节点上供应用程序使用的资源容器。在执行应用程序时，ApplicationMaster 监视容器直到完成。当应用程序完成时，ApplicationMaster 从 ResourceManager 注销其容器，执行周期就完成了。

图 11-2 显示了在 YARN 上运行的两个 Application，每个 Application 有一个 ApplicationMaster，如图中的 AM_1 和 AM_2。每个 ApplicationMaster 管理每个应用的每个具体任务，包括任务启动、任务监控、任务失败重启。图 11-2 显示了 AM_1 管理 3 个任务，具体包括 $Container_{1,1}$、$Container_{1,2}$、$Container_{1,3}$，AM_2 管理 4 个任务，具体包括 $Container_{2,1}$、$Container_{2,2}$、$Container_{2,3}$、$Container_{2,4}$。

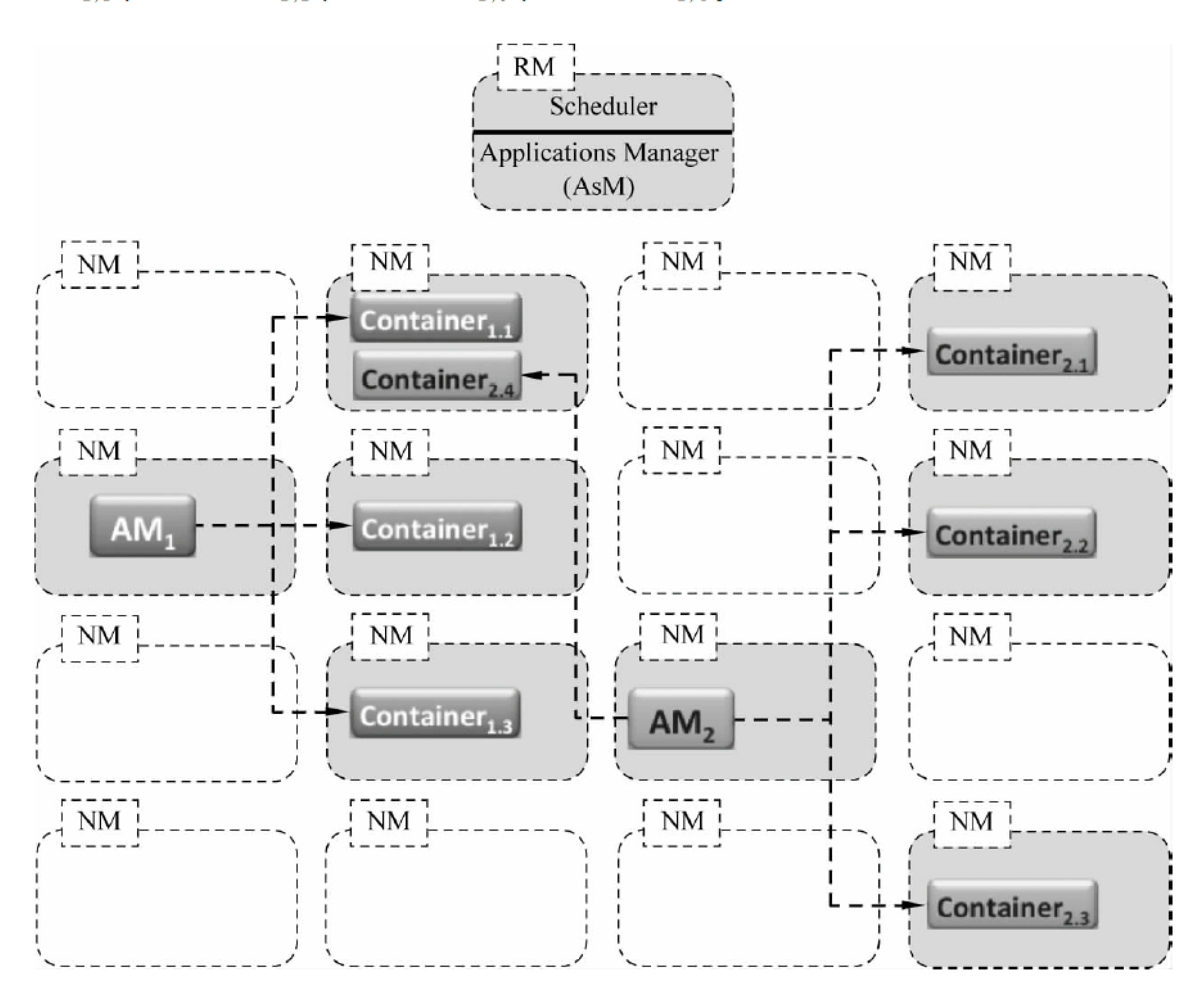


图 11-2 在 YARN 上运行 Application

下面从 YARN 资源分配模型和协议组件两部分来分析 YARN 的工作原理。

1. 资源分配模型

在早期的 Hadoop 版本中，每个集群中的节点资源被静态赋予具体的 slot 值，分为

Map slot 和 Reduce slot，这些 slot 无法在 Map 任务和 Reduce 任务之间共享。这种静态划分 Map slot 和 Reduce slot 的方式效果不佳，因为 MapReduce Job 运行会发生改变。实际情况是每个 MapReduce Job 随机提交，每一个都需要提交自己的 Map slot 和 Reduce slot 需求，这样很难使集群的资源利用达到最优。

目前解决这种问题的方式是利用 Container 的方法，这是一种更具有弹性的资源模型。资源请求以 Container 的方式发送，每个 Container 里面的属性都是非静态的。使用 Container 的方式只需要对 Container 中的每个属性定义一个最大值和一个最小值，比如为 memory 属性定义最大值和最小值。ApplicationMaster 请求容器 Container，设置对应的属性值，只需要最大值和最小值。

2. 协议组件

这里通过讲解 YARN 中 3 个重要的通信协议 protocol 来理解 YARN 的具体工作原理。

1）Client-ResourceManager

图 11-3 显示了一个 Application 在 YARN 上初始启动的过程，典型的是通过 Client 和 RM 通信来启动 Application。第 1 步，Client 发送启动 Application 请求给 RM 创建

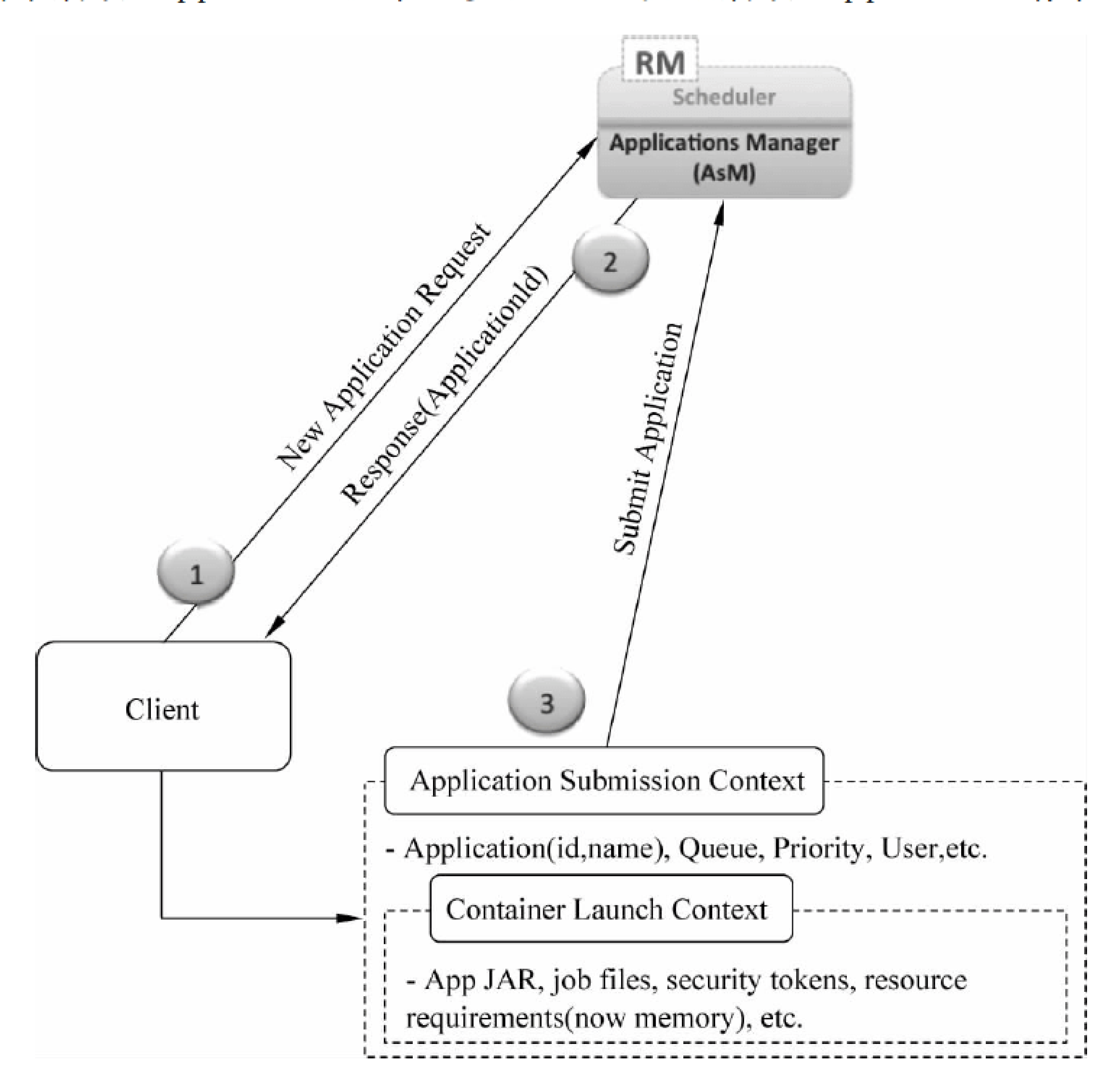


图 11-3 应用程序启动

一个新的 Application；第 2 步，RM 应答 Client 的请求，返回一个 ApplicationId 给 Client；第 3 步，在收到来自 RM 的响应后，Client 构建 Application Submission Context，信息包括 AppId、Queue、Priority 等，也包括 Container Launch Context，用于在 NM 上启动具体的 task。

Client 提交 Application Context 启动 Application 之后，可以发送 Application Report 查询请求给 RM 查询具体的 Application Report，RM 返回请求结果。如果中间出现其他问题，Client 可以取消删除该应用，如图 11-4 中的步骤 6 所示。

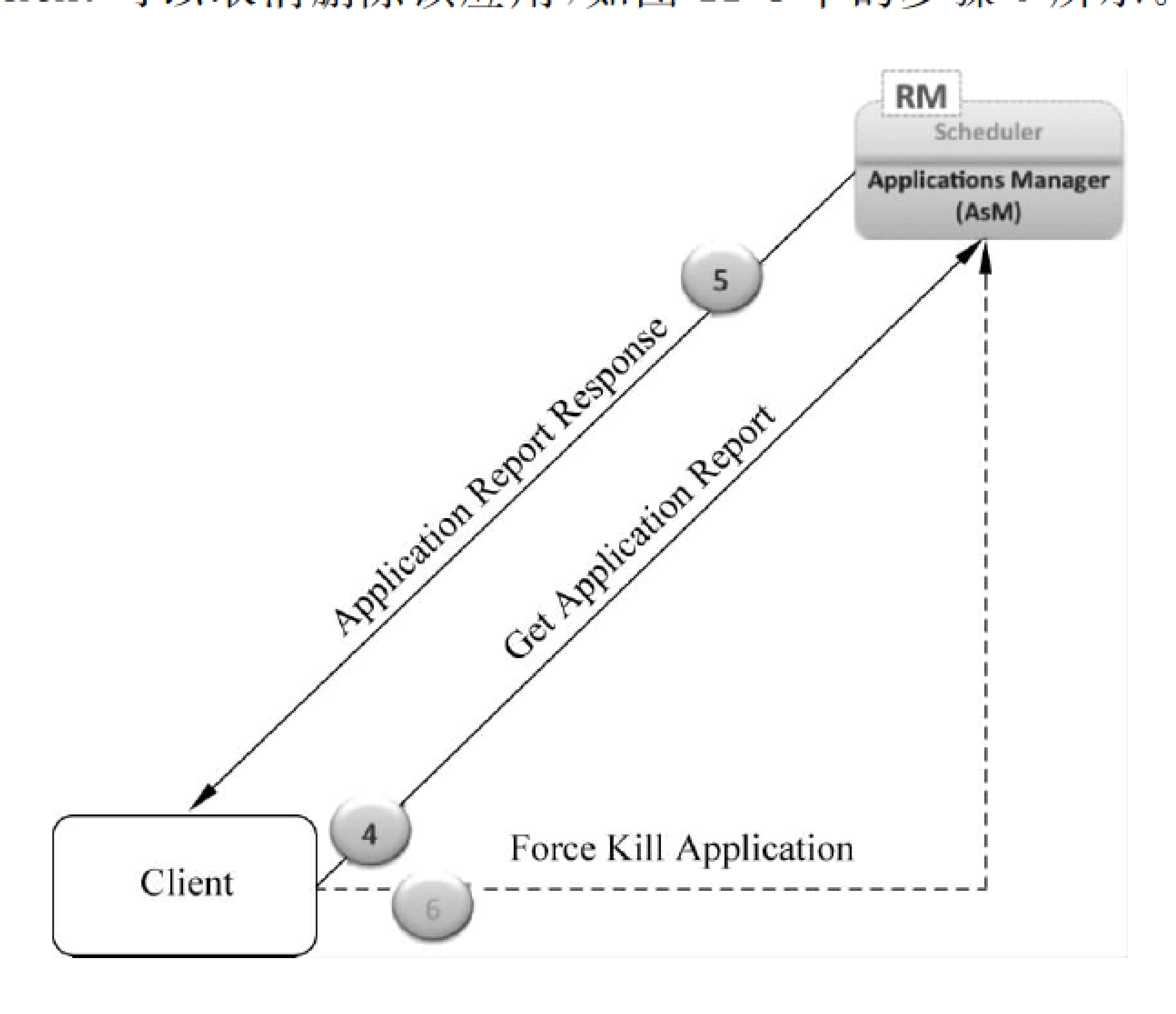


图 11-4　应用 Application 运行

2）ResourceManager － ApplicationMaster

当 RM 接受来自 Client 的 submission context 之后，寻找到一个具体有效的 Container 满足 AM 的需求，然后在具体的 NM 上启动 AM。图 11-5 描述了在 NM 上启动 AM 之后 AM 启动多个 task 的过程。第 1 步，AM 向 RM 登记，这个过程包括一个握手(handshake)过程以及发送 RPC 端口、tracking URL 等信息；第 2 步，RM 返回重要信息，包括当前集群的 min/max 资源容量(capacity)。AM 根据 min/max 资源容量(capacity)计算每个 task 的资源请求；第 3 步，发送具体的 Container 请求，同时也包括 AM 释放的 Container；第 5 步，RM 基于调度策略计算请求资源，返回满足要求的 Container；第 6 步，当完成 Application 时，AM 发送一个完成的消息给 RM。

3）ApplicationMaster-ContainerManager

图 11-6 显示了 AM(ApplicationMaster)与 NM(NodeManager)之间的通信。第 1 步，AM 根据 RM 返回的 Container 信息发送 Container Launch Context 到对应的 RM 上；当对应的 Container 运行时，AM 通过(2)和(3)获取对应的 Container 运行时信息。

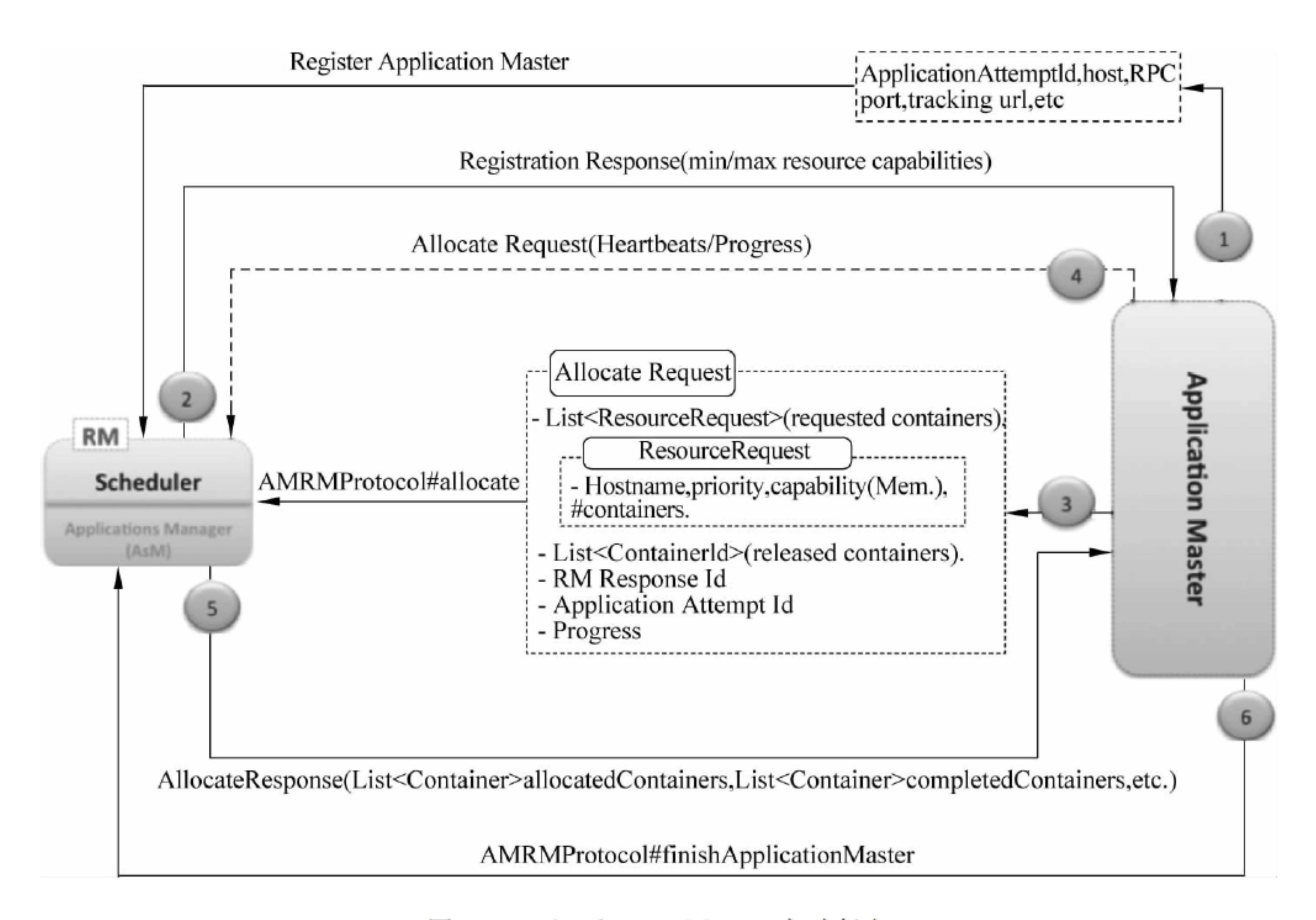


图 11-5　ApplicationMaster 启动任务

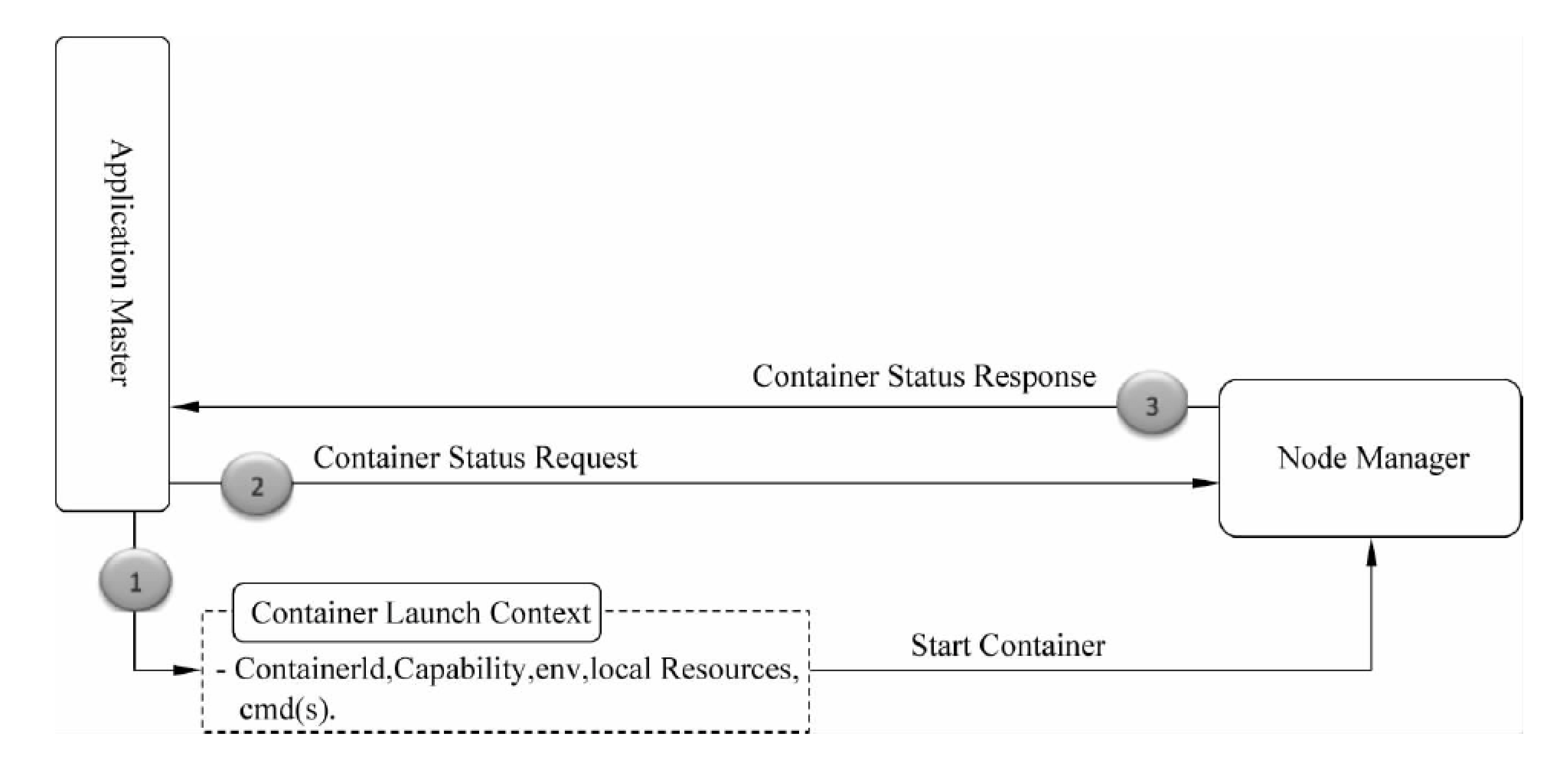


图 11-6　Application 监控任务运行

11.1.3　Apache Mesos

Mesos 是以与 Linux 内核同样的原则创建的，不同点仅在于抽象的层面。Mesos 内核运行在每一个机器上，同时通过 API 为各种应用提供跨数据中心和云的资源管理调度能力。这些应用包括 Hadoop、Spark、Kafka、Elastic Search。另外，Mesos 还可配合框架 Marathon 来管理大规模的 Docker 等容器化应用。

图 11-7 显示了 Mesos 的主要组成部分。Mesos 由一个 master daemon 来管理 Agent daemon 在每个集群节点上的运行，Mesos Applications（也称为 Frameworks）在这些 Agent 上运行 tasks。

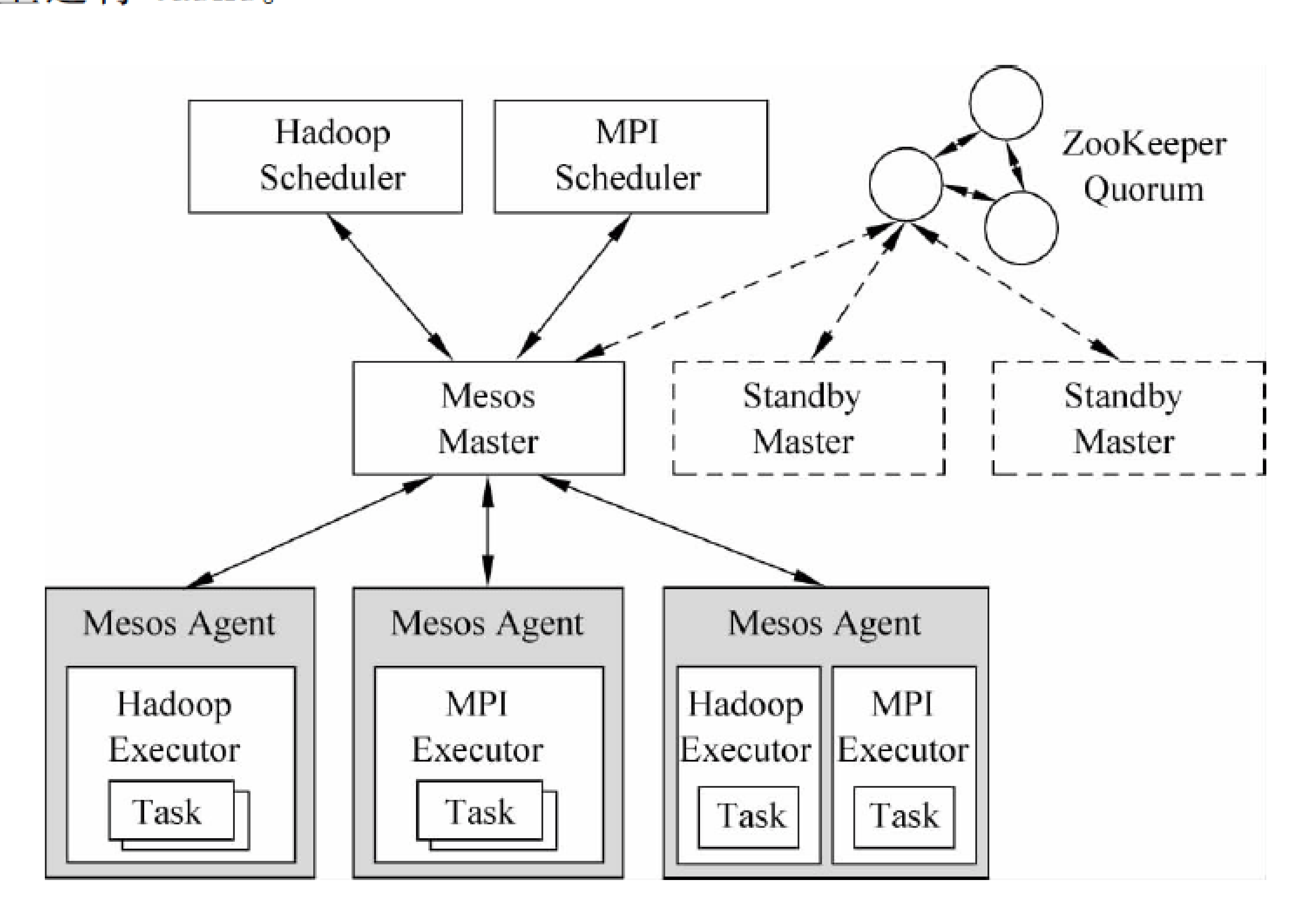


图 11-7　Mesos 架构图

Master 使用 Resource Offers 实现跨应用细粒度资源共享，如 CPU、内存、磁盘、网络等。Master 根据指定的策略来决定分配多少资源给计算框架，如公平共享策略或优先级策略。为了支持更多样性的策略，Master 采用模块化结构，这样就可以方便地通过插件形式来添加新的分配模块。

在 Mesos 上运行的计算框架由两个部分组成：一个是 Scheduler，通过注册到 Master 来获取集群资源；另一个是在 Agent 节点上运行的 Executor 进程，它可以执行计算框架的 Task。Master 决定为每个计算框架提供多少资源，通过计算框架的 Scheduler 来选择其中提供的资源。当计算框架同意了提供的资源时，它通过 Master 将 Task 发送到提供资源的 Agent 上运行。

图 11-8 是一个计算框架运行在 Mesos 上的资源供给流程，步骤如下：

(1) Agent1 向 Master 报告有 4 个 CPU 和 4GB 内存可用。

(2) Master 发送一个 Resource Offer 给 Framework1 来描述 Agent1 有多少可用资源。

(3) FrameWork1 中的 FW Scheduler 会答复 Master 有两个 Task 需要运行在 Agent1 上，一个 Task 需要< 2 cpu，1 gb 内存="">，另外一个 Task 需要< 1 cpu，2 gb 内存="">。

最后，Master 发送这些 Tasks 给 Agent1。之后，Agent1 还有一个 CPU 和 1GB 内存没有使用，所以分配模块可以把这些资源提供给 Framework2。

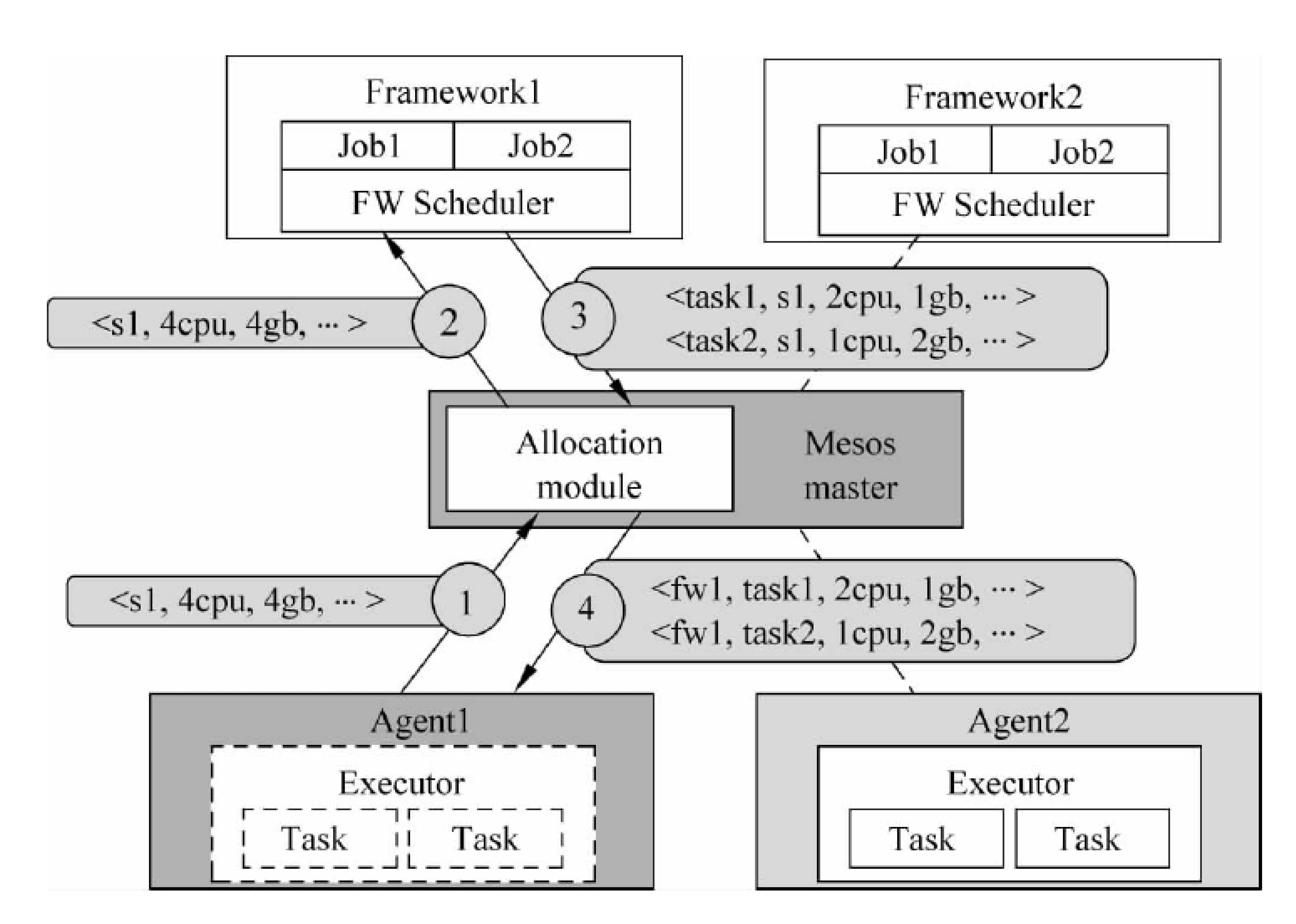


图 11-8　资源供给

11.1.4　Google Omega

Mesos、YARN 等集群管理系统采用的是双层调度器，相比独占调度器(Monolithic scheduler)具有更高的并发度，但是它具有以下缺点：

(1) 运行在这些集群管理系统上的计算框架无法知道整体集群的资源使用情况。

(2) 并发粒度小，采用的是悲观方式的并发控制(pessimistic concurrent control)。

针对上述双层调度器(two-level scheduler)的不足，Omega 设计了共享状态调度器(shared state scheduler)。该调度器将双层调度器中的集中式资源调度模块简化成了一些持久化的共享数据(状态)和针对这些数据的验证代码，而这里的“共享数据”实际上就是整个集群的实时资源使用信息。

11.2 资源管理模型

集群资源管理模型通常由两个部分组成，即资源表示模型和资源分配模型。由于这两个部分是耦合的，所以优化集群资源管理时需要同时结合这两个部分进行考虑。资源表示模型用于描述集群资源的组织方式，是集群资源统一管理的基础。从狭义上来讲，计算资源是指具有计算能力的资源，如 CPU 和 GPU 等。但实际上，对系统计算有影响的资源都可以划分到计算资源的范畴，包括内存、磁盘大小、I/O 和网络带宽等。合理的资源表示模型可以有效地利用资源，提高集群的利用率。

11.2.1 基于 slot 的资源表示模型

集群中每个节点的资源都是多维的，包括 CPU、内存、网络 I/O 和磁盘 I/O。为了简化资源管理问题，很多框架(如 Hadoop)引入“槽位”(slot)概念，并采用 slot 组织各个节点上的计算资源。实际上，基于 slot 的资源表示模型就是将各个节点上的资源等量切分成若干份，每一份用一个 slot 表示，同时规定任务可以根据实际需求占用多个 slot。通过引入“slot”这一概念，各个节点上的多维度资源被抽象成单一维度的 slot，这样可以把复杂的多维度资源分配问题转化成简单的 slot 分配问题，从而大大降低了资源管理问题的复杂度。

更进一步说，slot 相当于任务运行“许可证”。一个任务只有得到该“许可证”后才能获得运行的机会。这意味着每个节点上的 slot 数量决定了该节点上最大允许的任务并发度。同时为了区分不同任务所用资源量的差异，如 Hadoop 的作业被分为 Map Task 和 Reduce Task 两种类型，slot 则被分为 Map slot 和 Reduce slot 两种类型，并且只能分别被 Map Task 和 Reduce Task 使用。

11.2.2 基于最大最小公平原则的资源分配模型

对于任何共享集群的系统，资源分配都是一个至关重要的模块。一个最常用的分配策略是最大最小公平原则，其最早用于控制网络流量，以实现公平分配网络带宽。最大最小公平策略的基本含义是使得资源分配的最小分配量尽可能最大，它可以防止任何网络流被“饿死”，同时在一定程度上尽可能地增加每个流的速率。因此，最大最小公平策略被认为是一种很好的权衡有效性和公平性的自由分配策略，在经济、网络领域有着广

泛的应用，由其演变出来的加权最大最小公平模型被一些资源分配策略广泛地采用，如基于优先级、预留机制和限期的分配策略。最大最小公平模型同时也保证分配隔离，即用户确保接收自己的分配量而不用考虑其他用户的需求。

基于这些特点，大量的分配算法被提出来实现不同准确度的最大最小公平模型，例如轮询、均衡资源共享和加权公平队列等。这些算法被应用于各种各样的资源分配上，包括网络带宽、CPU、内存以及二级存储空间。但这些公平分配的工作主要集中在单一资源类型，同样，在多类型资源环境和需求异构化下，公平合理的分配策略也很重要。

为了支持多维度资源调度，越来越多的分配算法被提出，包括主资源公平调度算法，该算法扩展了最大最小公平算法，其能够在保证分配公平的前提下支持多维度资源的调度。在 DRF 算法中将所需份额（资源比例）最大的资源称为主资源，DRF 的基本设计思想则是将最大最小公平算法应用于主资源上，进而将多维资源调度问题转化为单维资源调度问题，即 DRF 总是最大化所有主资源占用量中最小的。由于 DRF 被证明非常适合应用于多资源和复杂需求的环境中，因此被越来越多的系统所采用，其中包括 Apache YARN 和 Apache Mesos。

11.3 资源调度策略

11.3.1 调度策略概述

在分布式计算领域中，资源分配问题实际上是一个任务调度问题。它的主要任务是根据当前集群中各个节点上的资源（包括 CPU、内存和网络）的剩余情况与各个用户作业的服务质量要求在资源和作业任务之间做出最优的匹配。由于用户对作业服务质量的要求是多样化的，分布式系统中的任务调度是一个多目标优化的问题。更进一步说，它是一个典型的 NP-hard 问题。

通常，分布式系统都会提供一个非常简单的调度机制——FIFO（First In First Out），即先来先服务。在该调度机制下，所有的用户作业都被提交到一个队列中，然后由调度器按照作业提交时间的先后顺序来选择将被执行的作业。但随着分布式计算框架的普及，集群的用户量越来越大，不同用户提交的应用程序往往具有不同的服务质量要求，典型的应用有以下 3 种：

（1）批处理作业。这种作业往往耗时较长，对完成时间一般没有严格要求，如数据挖掘、机器学习等方面的应用程序。

（2）交互式作业。这种作业期望能及时返回结果，如 SQL 查询（Hive）。

(3) 生产性作业。这种作业要求有一定量的资源保证,如统计值计算、垃圾数据分析等。

此外,不同应用程序对硬件资源的需求量也是不同的,如过滤/统计类作业一般为CPU密集型作业,而数据挖掘、机器学习的作业一般为I/O密集型作业。传统的FIFO调度算法虽然简单明了,但是它忽略了不同作业对资源的需求差异,严重时会影响作业的执行。因此,传统的FIFO调度策略不仅不能满足多样化需求,也不能充分利用硬件资源。

为了克服单队列FIFO调度器的不足,多种类型的多用户多队列调度器相继出现。这些调度策略允许管理员按照应用需求对用户或者应用程序分组,并为不同的分组分配不同的资源量,同时通过添加各种约束防止单个用户或应用程序独占资源,进而满足多样化的QoS需求。当前主要有两种多用户作业调度器的设计思路:第一种是在一个物理集群上虚拟多个子集群,典型的代表是HOD(Hadoop On Demand)调度器;另一种是扩展传统调度策略,使之支持多队列多用户,这样不同的队列拥有不同的资源量,可以运行不同的应用程序,典型的代表是Yahoo! 的Capacity Scheduler和Facebook的Fair Scheduler。

11.3.2 Capacity Scheduler 调度

Capacity Scheduler调度器是解决多用户情况下共享集群资源的调度方式,使每个提交的计算任务都可以在合理的时间内完成。

下面以Hadoop中的MapReduce作业为例来介绍Capacity Scheduler调度器。目前很多公司渐渐采用资源池的方式组织和管理资源,公司下属的多个部门机构如果需要使用资源则从总的资源池中分配具体配额。如果需要运行Hadoop MapReduce作业任务在这些共享集群资源上,则需要良好的资源调度方式。

Capacity Scheduler调度的思路如下:将总体的集群资源以可以预测和简单的方式划分到公司的多个子部门和机构,主要是以Job队列的方式;每个Job队列都有一个capacity的保证,也同时提供资源弹性功能,即一个队列未使用的资源可以给Job过载的队列使用;采用这种资源调度方式既可以提高系统的资源利用率,也可以确保所有Job的正常运行。举个简单的例子,假设建立了5个Job队列,则每个Job队列将会拥有20%的计算处理能力,用户当然可以自己定义Job应放到哪个Job队列中。

目前Hadoop中实现的Capacity Scheduler支持以下特性:

(1) 等级队列(Hierarchy Queue)。采用等级队列的方式可以确保所有的空闲资源在所有用户中共享,提高资源的控制能力。

(2) 容量保证(Capacity Guarantee)。每个队列都有一定比例的资源。

(3) 安全保证(Security Guarantee)。每个队列都有 ACL,限制可以存放的用户 Job。

(4) 弹性(Elasticity)。分配给队列的空闲资源超过它的容量,多余的空闲资源可以分配给其他队列。

(5) 多用户(Multi-tenancy)。提供给每个应用、用户和队列的资源是有限制的,防止它们独占整体的队列资源和集群资源。

(6) 可操作。运行时配置(可以在运行时更改配置)和 Drain Applicatin(系统管理员可以停止队列直到现有的应用完成才允许新的 Job 添加到队列中)。

(7) 基于资源的调度。支持资源密集型应用,这些应用可以指定高于默认的资源需求,同时协调不同的资源需求。

表 11-1 是 Hadoop 中 Capacity Scheduler 调度器的配置。

表 11-1 conf/yarn-site.xml 配置

Property	Value
yarn.resourcemanager.scheduler.class	org.apache.hadoop.yarn.server.resourcemanager.scheduler.capacity.CapacityScheduler

下面是具体的队列配置:

```
<property>
  <name>yarn.scheduler.capacity.root.queues</name>
  <value>a,b,c</value>
  <description>The queues at the this level (root is the root queue).
  </description>
</property>

<property>
  <name>yarn.scheduler.capacity.root.a.queues</name>
  <value>a1,a2</value>
  <description>The queues at the this level (root is the root queue).
  </description>
</property>

<property>
  <name>yarn.scheduler.capacity.root.b.queues</name>
  <value>b1,b2,b3</value>
  <description>The queues at the this level (root is the root queue).
  </description>
</property>
```

11.3.3 Fair Scheduler 调度

公平调度是一种赋予作业(Job)资源的方法,它的目的是让所有作业随着时间的推移都能平均地获取等同的共享资源。当单独一个作业运行时,它将使用整个集群。当有其他作业被提交上来时,系统会将任务(task)空闲时间片(slot)赋给这些新的作业,以使每一个作业大概获取到等量的CPU时间。与Hadoop默认调度器维护一个作业队列不同,这个特性让小作业在合理的时间内完成的同时又不"饿"到消耗较长时间的大作业。它也是一个在多用户间共享集群的简单方法。公平共享可以和作业优先权搭配使用——优先权像权重一样用作决定每个作业所能获取的整体计算时间的比例。

公平调度器按资源池(pool)来组织作业,并把资源公平地分到这些资源池里。默认情况下,每一个用户拥有一个独立的资源池,以使每个用户都能获得一份等同的集群资源而不管他们提交了多少作业。按用户的UNIX群组或作业配置(JobConf)属性来设置作业的资源池也是可以的。在每一个资源池内会使用公平共享(fair sharing)的方法在运行作业之间共享容量(capacity)。用户也可以给予资源池相应的权重,以不按比例的方式共享集群。

除了提供公平共享方法外,公平调度器还提供了资源池中最小使用资源保证,这种方式在特定场合和生产环境下可以起到很有效的作用。当一个资源池包含作业时,它至少能获取到它的最小共享资源,但是当资源池不完全需要它所拥有的保证共享资源时,额外的部分会在其他资源池间进行切分。

在常规操作中,当提交了一个新作业时,公平调度器会等待已运行作业中的任务完成以释放时间片给新的作业。但是公平调度器也支持在可配置的超时时间后对运行中的作业进行抢占。如果新的作业在一定时间内还获取不到最小的共享资源,这个作业被允许去终结已运行作业中的任务以获取运行所需要的资源。因此抢占可以用来保证"生产"作业在指定时间内运行的同时也让Hadoop集群能被实验或研究作业使用。另外,作业的资源在可配置的超时时间(一般设置大于最小共享资源超时时间)内拥有不到其公平共享资源(fair sharing)一半的时候也允许对任务进行抢占。在选择需要结束的任务时,公平调度器会在所有作业中选择那些最近运行起来的任务,以最小化被浪费的计算。抢占不会导致被抢占的作业失败,因为Hadoop作业能"容忍"丢失任务,这只是会让它们的运行时间更长。

最后,公平调度器还可以限制每个用户和每个资源池的并发运行作业数量。当一个用户必须一次性提交数百个作业时,或当大量作业并发执行时,用来确保中间数据不会

塞满集群上的磁盘空间，这是很有用的。设置作业限制会使超出限制的作业被列入调度器的队列中进行等待，直到一些用户/资源池的早期作业运行完毕。系统会根据作业优先权和提交时间的排列来运行每个用户/资源池中的作业。

以下是 Hadoop 中配置 Fair Scheduler 的方式：

```
<property>
    <name>mapred.jobtracker.taskScheduler</name>
    <value>org.apache.hadoop.mapred.FairScheduler</value>
</property>
```

实现公平调度分两个方面：计算每个作业的公平共享资源，以及当一个任务的时间片可用时选择哪个作业去运行。

在选择了运行作业之后，调度器会跟踪每一个作业的“缺额”——作业在理想调度器上所应得的计算时间与实际所获得的计算时间的差额。这是一个测量作业的“不公平”待遇的度量标准。每过几百毫秒，调度器就会通过查看各个作业在这个间隔内运行的任务数与它的公平共享资源的差额来更新各个作业的缺额。当有任务时间片可用时，它会被赋给拥有最高缺额的作业。但有一个例外——如果有一个或多个作业都没有达到它们的资源池容量的保证量，将只在这些“贫穷”的作业间进行选择（再次基于它们的缺额），以保证调度器能尽快地满足资源池的保证量。

公平共享资源是依据各个作业的“权重”通过在可运行作业之间平分集群容量计算出来的。默认权重是基于作业优先权的，每一级优先权的权重是下一级的两倍（例如，VERY_HIGH 的权重是 NORMAL 的 4 倍）。但是，权重也可以基于作业的大小和年龄。对于在一个资源池内的作业，公平共享资源还会考虑这个资源池的最小保证量，接着再根据作业的权重在这个资源池内的作业间划分这个容量。

在用户或资源池的运行作业限制没有达到上限的时候，做法和标准的 Hadoop 调度器一样，在选择要运行的作业时，首先根据作业的优先权对所有作业进行排序，然后再根据提交时间进行排序。对于上述排序队列中超出用户/资源池限制的作业将会被排队并等待空闲时间片，直到它们可以运行。在这段时间内，它们被公平共享计算忽略，不会获得或失去缺额（它们的公平均分量被设为 0）。

抢占是定期检查是否有作业的资源低于其最小共享资源或低于其公平共享资源的一半。如果一个作业的资源低于其共享资源的时间足够长，它将被允许去结束其他作业的任务。所选择的任务是所有作业中最近运行起来的任务，以最小化被浪费的计算。

11.4 在 YARN 上运行计算框架

11.4.1 MapReduce on YARN

由于 MapReduce 的 JobTracker/TaskTracker 机制需要通过大规模的调整来修复它在可扩展性、内存消耗、线程模型、可靠性等上的缺陷，为从根本上解决旧 MapReduce 框架的性能瓶颈，MapReduce 框架需要完全重构，图 11-9 是新的 YARN 系统架构图。重构的根本思想是将 JobTracker 的两个主要功能分离成单独的组件，这两个功能是资源管理和任务调度/监控。新的资源管理器全局管理所有应用程序计算资源的分配，每一个应用的 ApplicationMaster 负责相应的调度和协调。ResourceManager 和每一台机器的节点管理服务器管理用户在这台机器上的进程并能对计算进行组织。

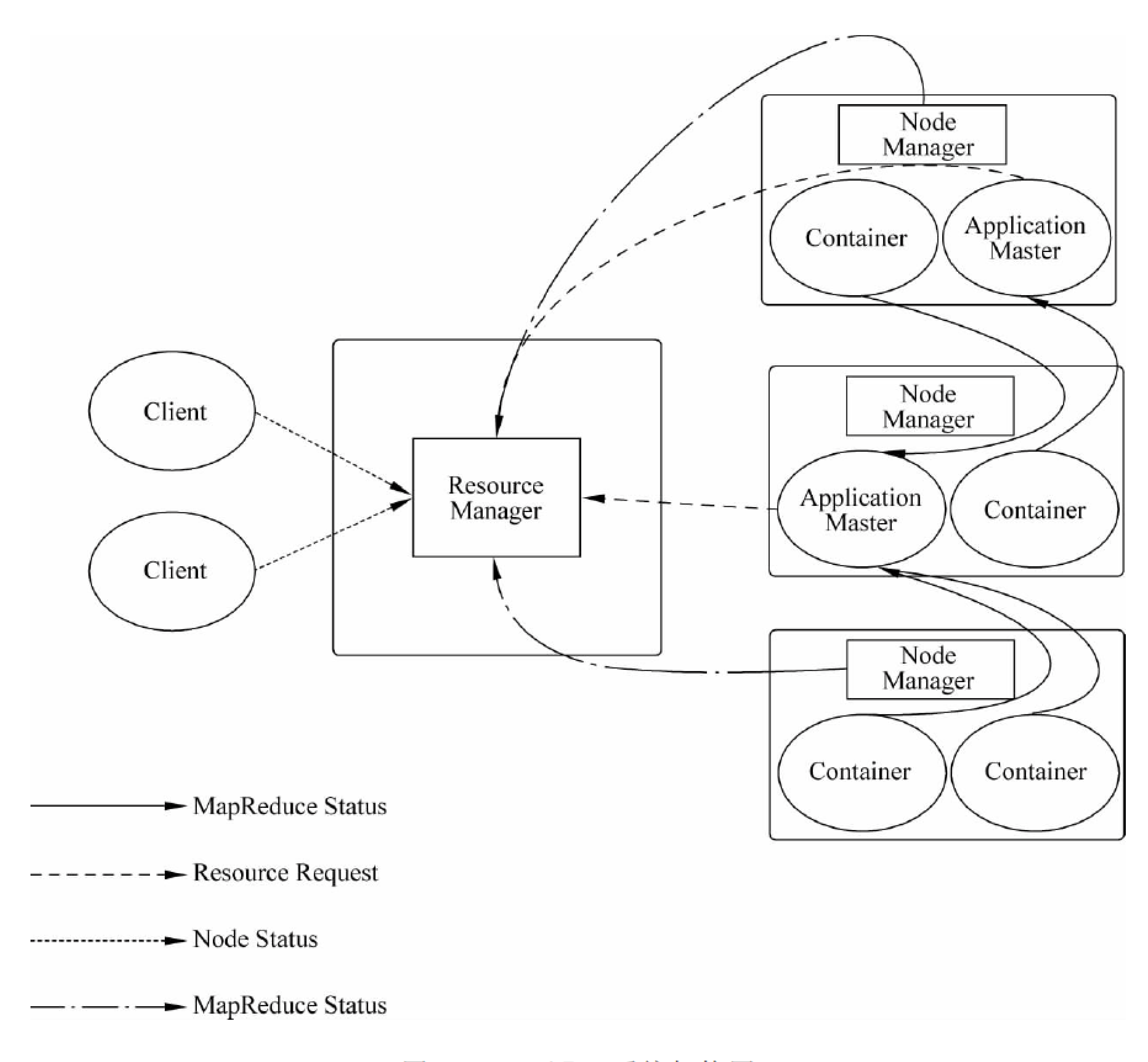


图 11-9　YARN 系统架构图

事实上，每一个应用的 ApplicationMaster 是一个详细的框架库，它结合从 ResourceManager 获得的资源和 NodeManager 协同工作来运行和监控任务。在图 11-9 中，ResourceManager 支持分层级的应用队列，这些队列享有集群一定比例的资源。它就是一个调度器，在执行过程中不对应用进行监控和状态跟踪。同样，它也不能重启因应用失败或者硬件错误而运行失败的任务。

ResourceManager 是基于应用程序对资源的需求进行调度的，每一个应用程序需要不同类型的资源，因此就需要不同的容器。资源包括内存、CPU、磁盘、网络等。资源管理器提供调度策略，它负责将集群资源分配给多个队列和应用程序。调度器可以基于现有的能力进行调度。

在图 11-9 中，NodeManager 是每一台机器框架的代理，是执行应用程序的容器，监控应用程序的资源使用情况，如 CPU、内存、硬盘、网络等，并且向调度器汇报。

每一个应用程序的 ApplicationMaster 的职责有向调度器索要适当的资源容器、运行任务、跟踪应用程序的状态、监控进程、处理任务的失败原因等。

11.4.2 Spark on YARN

Spark 是类 Hadoop MapReduce 的通用并行框架。Spark 拥有 Hadoop MapReduce 所具有的优点，不同于 MapReduce 的是 Job 中间输出结果可以保存在内存中，从而不再需要读/写磁盘，因此 Spark 能更好地适用于数据挖掘与机器学习等需要迭代的 MapReduce 的算法。Spark 在 YARN 中有 yarn-cluster 和 yarn-client 两种运行模式。

1. yarn-cluster

Spark Driver 首先作为一个 ApplicationMaster 在 YARN 集群中启动，客户端提交给 ResourceManager 的每一个 Job 都会在集群的 worker 节点上分配一个唯一的 ApplicationMaster，由该 ApplicationMaster 管理全生命周期的应用。因为 Driver 程序在 YARN 中运行，所以事先不用启动 Spark Master/Client，应用的运行结果不能在客户端显示(可以在 History Server 中查看)，所以最好将结果保存在 HDFS 中，客户端的终端显示的是作为 YARN 的 Job 的简单运行状况。

yarn-cluster 的运行步骤如下：

(1) 由 Client 向 ResourceManager 提交请求，并上传 JAR 到 HDFS 上。

(2) ResouceManager 向 NodeManager 申请资源，创建 Spark ApplicationMaster。

(3) NodeManager 启动 Spark App Master 并注册。

(4) Spark ApplicationMaster 从 HDFS 中找到 JAR 文件，启动 DAGscheduler 和

YARN Cluster Scheduler。

(5) ResourceManager 注册申请 Container 资源。

(6) ResourceManager 通知 NodeManager 分配 Container。

(7) Spark ApplicationMaster 和 Container 进行交互,完成这个分布式任务。

2. yarn-client

在 yarn-client 模式下,Driver 运行在 Client 上,通过 ApplicationMaster 向 RM 获取资源。本地 Driver 负责与所有的 Executor Container 进行交互,并将最后的结果汇总。结束终端,相当于关闭这个 spark 应用。客户端的 Driver 将应用提交给 YARN 后,YARN 会先后启动 ApplicationMaster 和 Executor。

ApplicationMaster 和 Executor 都是装载在 Container 里运行的,ApplicationMaster 分配的内存是 driver-memory,Executor 分配的内存是 executor-memory。同时,因为 Driver 在客户端,所以程序的运行结果可以在客户端显示,Driver 以进程名为 SparkSubmit 的形式存在。

11.4.3 YARN 程序设计

YARN 是一个资源管理系统,负责集群资源的管理和分配。如果想将一个新的应用程序运行在 YARN 之上,通常需要编写两个组件,即 Client 和 ApplicationMaster。在实际应用中专业的开发人员编写这两个组件,并提供给上层的应用程序用户使用。如果大量应用程序可抽象成一种通用框架,只需实现一个 Client 和一个 ApplicationMaster,然后让所有应用程序重用这两个组件即可。

通常,编写一个 YARN Application 涉及下面 3 个 PRC 协议:

(1) ClientRMProtocol。Client 通过该协议将应用程序提交给 ResourceManager 查询应用程序的运行状态、杀死应用程序等。

(2) AMRMProtocol。ApplicationMaster 使用该协议向 ResourceManager 注册、申请资源以运行自己的各个任务。

(3) ContainerManager。ApplicationMaster 使用该协议要求 NodeManager 启动/撤销 Container,或者获取各个 Container 的运行状态。

编写 YARN 程序的步骤如下:

1. 编写 Client

客户端通常只需要与 ResourceManager 交互,具体如下:

(1) 获取 Application Id。客户端通过 RPC 协议 ClientRMProtocol 向 ResourceManager 发送应用程序提交请求——GetNewApplicationRequest，ResourceManager 返回 GetNewApplicationResponse。

(2) 提交 ApplicationMaster。将启动 ApplicationMaster 所需的全部信息打包到数据结构 ApplicationSubmissionContext 中，所需信息主要包括 Application Id、Application Name、Application Priority、Application 所属队列、Application 启动用户名、Application 对应的 Container 信息。客户端调用 ClientRMProtocol#submitApplication(ApplicationSubmissionContext)将 ApplicationMaster 提交到 ResourceManager 上。ResourceManager 收到请求后会为 ApplicationMaster 寻找合适的节点，并在该节点上启动它。

2. 编写 ApplicationMaster

ApplicationMaster 需要与 ResoureManager 和 NodeManager 交互，具体步骤如下：

(1) 注册。ApplicationMaster 首先需要通过 RPC 协议 AMRMProtocol 向 ResourceManager 发送注册请求——RegisterApplicationMasterRequest，该数据结构中包含 ApplicationMaster 所在节点的 host、RPC port 和 TrackingUrl 等信息，而 ResourceManager 将返回 RegisterApplicationMasterResponse，该数据结构中包含多种信息，包括该应用程序的 ACL 列表、资源可使用上限和下限等。

(2) 申请资源。根据每个任务的资源需求，ApplicationMaster 向 ResourceManager 申请一系列用于运行任务的 Container，ApplicationMaster 使用 ResourceRequest 类描述每个 Container，一旦为任务构造了 Container，ApplicationMaster 就会使用 RPC 函数 AMRMProtocol#allocate 向 ResourceManager 发送一个 AllocateRequest 对象，以请求分配这些 Container，ResourceManager 会为 ApplicationMaster 返回一个 AllocateResponse 对象，该对象中的主要信息包含在 AMResponse 中，ApplicationMaster 会不断追踪已经获取的 Container，且只有当需求发生变化时才允许重新为 Container 申请资源。

(3) 启动 Container。当 ApplicationMaster 从 ResourceManager 收到新分配的 Container 列表后，使用 RPC 函数 ContainerManager#startContainer 向对应 NodeManager 发送 ContainerLaunchContext 以启动 Container。

ApplicationMaster 不断重复步骤(2)和步骤(3)，直到所有任务运行成功，它会调用 AMRMProtocol#finishApplicationMaster，以告诉 ResourceManager 自己运行结束。

下面是一个运行在 YARN 上的简单程序。设定场景是在 YARN 上启动一个 shell 命令，启动的 AM(ApplicationMaster)是一个不被管理的 AM，采用上面描述的 YARN 程序启动步骤编写。

首先初始化一个 YARN Client：

```
public boolean init(String[] args) throws ParseException {

    Options opts = new Options();
    opts.addOption("appname", true,
            "Application Name. Default value - UnmanagedAM");
    opts.addOption("priority", true, "Application Priority. Default 0");
    opts.addOption("queue", true,
            "RM Queue in which this application is to be submitted");
    opts.addOption("master_memory", true,
        "Amount of memory in MB to be requested to run the"
        "application master");
    opts.addOption("cmd", true, "command to start unmanaged"
        "AM (required)");
    opts.addOption("classpath", true, "additional classpath");
    opts.addOption("help", false, "Print usage");
    CommandLine cliParser = new GnuParser().parse(opts, args);

    if (args.length == 0) {
        printUsage(opts);
        throw new IllegalArgumentException(
            "No args specified for client to initialize");
    }

    if (cliParser.hasOption("help")) {
        printUsage(opts);
        return false;
    }

    appName = cliParser.getOptionValue("appname", "UnmanagedAM");
    amPriority = Integer.parseInt(cliParser.getOptionValue("priority", "0"));
    amQueue = cliParser.getOptionValue("queue", "default");
    classpath = cliParser.getOptionValue("classpath", null);

    amCmd = cliParser.getOptionValue("cmd");
    if (amCmd == null) {
        printUsage(opts);
        throw new IllegalArgumentException(
            "No cmd specified for application master");
    }
```

```
        YarnConfiguration yarnConf = new YarnConfiguration(conf);
        rmClient = YarnClient.createYarnClient();
        rmClient.init(yarnConf);

        return true;
    }
```

在 YARN 上启动一个 AM(ApplicationMaster)：

```
    public void launchAM(ApplicationAttemptId attemptId)
        throws IOException, YarnException {
        Credentials credentials = new Credentials();
        Token<AMRMTokenIdentifier> token =
            rmClient.getAMRMToken(attemptId.getApplicationId());
        //服务将是空的,但没关系
        //我们只是将 AMRMToken 传递给真正的 AM
        //最终设置正确的服务地址
        credentials.addToken(token.getService(), token);
        File tokenFile = File.createTempFile("unmanagedAMRMToken","",
            new File(System.getProperty("user.dir")));
        try {
          FileUtil.chmod(tokenFile.getAbsolutePath(), "600");
        } catch (InterruptedException ex) {
          throw new RuntimeException(ex);
        }
        tokenFile.deleteOnExit();
        DataOutputStream os = new DataOutputStream(new FileOutputStream(tokenFile,
            true));
        credentials.writeTokenStorageToStream(os);
        os.close();

        Map<String, String> env = System.getenv();
        ArrayList<String> envAMList = new ArrayList<String>();
        boolean setClasspath = false;
        for (Map.Entry<String, String> entry : env.entrySet()) {
          String key = entry.getKey();
          String value = entry.getValue();
          if(key.equals("CLASSPATH")) {
            setClasspath = true;
            if(classpath != null) {
```

```
            value = value + File.pathSeparator + classpath;
          }
        }
        envAMList.add(key + " = " + value);
      }

      if(!setClasspath && classpath!= null) {
        envAMList.add("CLASSPATH = " + classpath);
      }
    ContainerId containerId = ContainerId.newContainerId(attemptId, 0);

    String hostname = InetAddress.getLocalHost().getHostName();
    envAMList.add(Environment.CONTAINER_ID.name() + " = " + containerId);
    envAMList.add(Environment.NM_HOST.name() + " = " + hostname);
    envAMList.add(Environment.NM_HTTP_PORT.name() + " = 0");
    envAMList.add(Environment.NM_PORT.name() + " = 0");
    envAMList.add(Environment.LOCAL_DIRS.name() + " = /tmp");
    envAMList.add(ApplicationConstants.APP_SUBMIT_TIME_ENV + " = "
          + System.currentTimeMillis());
    envAMList.add ( ApplicationConstants. CONTAINER _TOKEN _FILE _ENV _NAME + " = " +
  tokenFile.getAbsolutePath());

    String[] envAM = new String[envAMList.size()];
    Process amProc = Runtime.getRuntime().exec(amCmd, envAMList.toArray(envAM));

    final BufferedReader errReader =
          new BufferedReader(new InputStreamReader(
              amProc.getErrorStream(), Charset.forName("UTF-8")));
    final BufferedReader inReader =
          new BufferedReader(new InputStreamReader(
              amProc.getInputStream(), Charset.forName("UTF-8")));

    //读取错误输入流
    //因为这将释放错误流缓冲区
    Thread errThread = new Thread() {
        @Override
        public void run() {
          try {
            String line = errReader.readLine();
            while((line != null) && !isInterrupted()) {
              System.err.println(line);
              line = errReader.readLine();
```

```
          }
        } catch(IOException ioe) {
          LOG.warn("Error reading the error stream", ioe);
        }
      }
    };
    Thread outThread = new Thread() {
      @Override
      public void run() {
        try {
          String line = inReader.readLine();
          while((line != null) && !isInterrupted()) {
            System.out.println(line);
            line = inReader.readLine();
          }
        } catch(IOException ioe) {
          LOG.warn("Error reading the out stream", ioe);
        }
      }
    };
    try {
      errThread.start();
      outThread.start();
    } catch (IllegalStateException ise) { }

    //等待进程完成,并检查退出代码
    try {
      int exitCode = amProc.waitFor();
      LOG.info("AM process exited with value: " + exitCode);
    } catch (InterruptedException e) {
      e.printStackTrace();
    } finally {
      amCompleted = true;
    }

    try {
      //确保错误线程在 Windows 上退出
      //这些线程有时会卡住并挂起执行超时
      //然后在销毁过程后加入
      errThread.join();
      outThread.join();
      errReader.close();
```

```
            inReader.close();
          } catch (InterruptedException ie) {
            LOG.info("ShellExecutor: Interrupted while reading the error/out stream", ie);
          } catch (IOException ioe) {
            LOG.warn("Error while closing the error/out stream", ioe);
          }
          amProc.destroy();
        }
```

在终端运行以下命令：

```
bin/hadoop jar /home/qzhong/yarn-0.0.1.jar alibook.yarn.UnmanagedAMLauncher -cmd "cat
/etc/hosts"
```

上面的 yarn-0.0.1.jar 为 Maven 编译产生的 JAR 包文件，cmd 参数为需要执行的命令参数。图 11-10 所示为运行结果。

```
16/11/22 03:01:24 INFO yarn.UnmanagedAMLauncher: Initializing Client
16/11/22 03:01:25 INFO yarn.UnmanagedAMLauncher: Starting Client
16/11/22 03:01:25 INFO client.RMProxy: Connecting to ResourceManager at dell122/10.61.2.122:8032
16/11/22 03:01:25 INFO yarn.UnmanagedAMLauncher: Setting up application submission context for ASM
16/11/22 03:01:25 INFO yarn.UnmanagedAMLauncher: Setting unmanaged AM
16/11/22 03:01:25 INFO yarn.UnmanagedAMLauncher: Submitting application to ASM
16/11/22 03:01:25 INFO impl.YarnClientImpl: Submitted application application_1477880581089_0026
16/11/22 03:01:26 INFO yarn.UnmanagedAMLauncher: Got application report from ASM for, appId=26, appAttemptId=app
477880581089_0026_000001, clientToAMToken=null, appDiagnostics=, appMasterHost=N/A, appQueue=default, appMasterR
, appStartTime=1479754885486, yarnAppState=ACCEPTED, distributedFinalState=UNDEFINED, appTrackingUrl=N/A, appUse
16/11/22 03:01:26 INFO yarn.UnmanagedAMLauncher: Launching AM with application attempt id appattempt_14778805810
00001
16/11/22 03:01:26 INFO yarn.UnmanagedAMLauncher: AM process exited with value: 0
127.0.0.1    localhost localhost4 localhost4.localdomain4
::1          localhost localhost6 localhost6.localdomain6
```

图 11-10　YARN 应用的运行结果

11.5　阿里云伏羲调度系统

“飞天”是阿里巴巴的云计算平台，其中的分布式调度系统被命名为“伏羲”（代码名称 Fuxi），该名字来自我国古代神话人物。伏羲主要负责管理集群的机器资源和调度并发的计算任务，目前支持离线数据处理（DAG Job）和在线服务（Online Service），为上层分布式应用（如 MaxCompute/OSS/OTS）提供稳定、高效、安全的资源管理和任务调度服务，为阿里巴巴集团打造数据分享第一平台的目标提供了强大的计算引擎。

11.5.1　伏羲调度系统架构

伏羲系统在设计上采用 Master/Slave 架构，如图 11-11 所示，该系统有一个被称为

“伏羲 Master”的集群控制中心，其余每台机器上会运行一个叫“伏羲 Agent”的守护进程，守护进程除了管理节点上运行的任务以外，还负责收集该节点上资源的使用情况，并将之汇报给控制中心。控制中心与伏羲 Agent 之间使用心跳机制，以监测节点的健康状态。当用户向伏羲 Master 提交一个任务时，伏羲 Master 会调度出一个可用节点在其上启动任务的主控进程 AppMaster，主控进程随后会向伏羲 Master 提出资源请求，得到伏羲 Master 分配的资源后，AppMaster 通知相应节点上的伏羲 Agent 开始运行任务 Worker。伏羲是一个支持多任务并发的调度系统，控制中心——伏羲 Master 负责在多个任务之间仲裁，支持优先级、资源 Quota 配额和抢占。

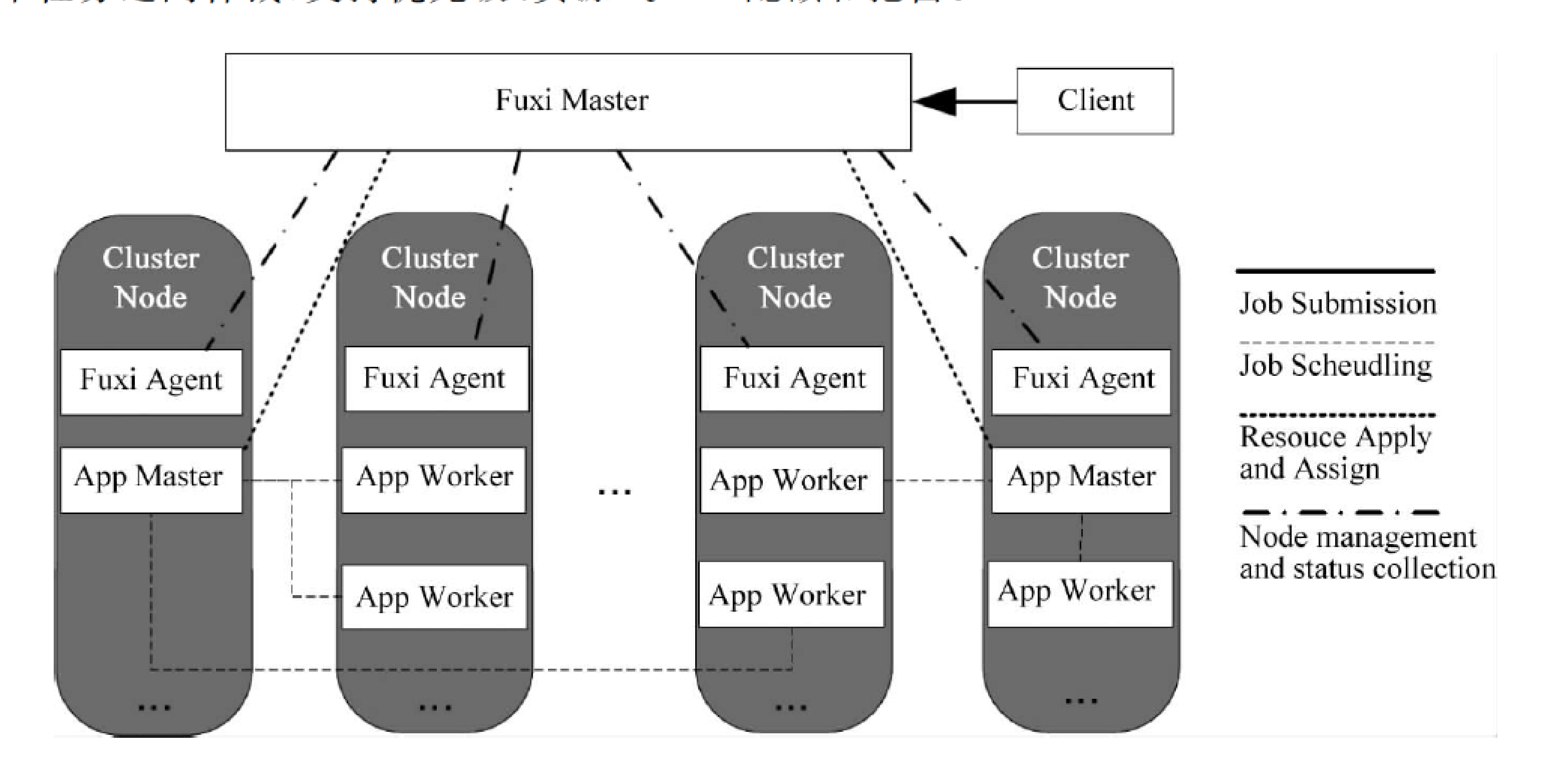


图 11-11 伏羲调度系统架构

使用伏羲，用户可以运行常见的 MapReduce 任务，还可以托管在线服务，满足不同应用场景的需求。多用户可以共享集群，伏羲支持配置分组的资源配额，限定每个用户组可以使用的计算资源，对于紧急任务(如重要数据报表)可以提高任务优先级来优先使用计算资源。

11.5.2 5K 挑战

伏羲是一个可扩展的分布式调度系统，将伏羲调度系统扩展到集群 5K(5000)台规模具有很大的挑战性，中间会碰到很多陷阱，原因主要在 3 个方面：规模放大效应，当系统扩展到数千节点时，原本非瓶颈与规模成正比的环节，其影响会被放大；木桶效应，在很多时候，系统中 99%的地方都被优化过，完成剩下 1%的优化看起来只是“锦上添花”，然而这 1%很可能就会成为影响系统性能的致命的瓶颈；长路径模块依赖，有些请求处理过程可能需要跨越多个模块(包括外部模块)，而外部模块性能的不稳定性最终可能会

影响到这个请求的处理性能和稳定性。

伏羲系统扩展到5K需要多方面综合考虑，这给伏羲系统带来规模、性能、稳定、运维等多个方面的技术挑战：

（1）通信消息DDoS。在5000规模的集群中，不同进程之间的RPC请求数量会随规模猛增，网络中的总请求数可达10 000 QPS，极易造成系统中单点进程的消息拥塞，从而导致请求处理严重超时。另外，消息处理还存在队头阻塞（HoL）问题。

（2）关键函数OPS。伏羲Master是资源调度的中心节点，内部关键调度函数的OPS必须达到极高的标准，否则可能会因为木桶效应影响到集群整体的调度性能。

（3）故障恢复对外部模块依赖。伏羲Master具有对用户透明的故障恢复功能（Failover），其恢复过程依赖写在女娲（女娲是飞天平台的协同系统，如名字服务）的Checkpoint上。因此，其整体恢复速度会受到女娲访问速度的影响。

针对上述伏羲5K集群规模背景，采取大量的优化工作来规避可能的问题以及陷阱，具体设计到架构设计、实现细节以及模块依赖。针对每个具体的细节，从底层工作原理深入优化，找到具体的性能瓶颈。

11.5.3 伏羲优化实践

在模块依赖性能优化方面，伏羲Master支持故障恢复，在重启后进行故障恢复时需要从女娲读取所有任务的描述文件（Checkpoint），以继续运行用户任务。考虑到之前女娲服务在服务器端对文件内容没有做持久化，伏羲Master在读取了Checkpoint后还会再写一次女娲，这个回写操作性能依赖于女娲模块。在5000节点的集群上，名字解析压力的显著增加导致女娲在Server的回写操作上也出现了性能下降问题，最终通过模块依赖传递到了伏羲Master，从而影响了故障恢复的性能。经测试观察，一次Checkpoint回写就消耗70s，这大大降低了伏羲系统的可用性。

所以需要对伏羲Master故障恢复进行优化。从伏羲Master的角度而言，在故障恢复时刚读取的Checkpoint内容在女娲服务器端是不会发生改变的，因此读取Checkpoint后没有必要回写到服务器端，只需要通知本地的女娲Agent让其代理即可，Agent会负责服务器宕机重启时向服务器推送本地缓存的文件内容。于是和女娲团队的同学合作，在女娲API中新增加一个只写本地的接口，这样伏羲Master规避了在故障恢复时回写Checkpoint的性能风险。优化后，在5000节点集群和并发5000任务的测试规模下，一次故障恢复中处理Checkpoint操作仅需18s（主要时间在一次读取）。可见在分布式系统中对外部模块的依赖哪怕只是一个RPC请求也可能是“性能陷阱”，在设计和实现时应尽量避免出现在关键路径上。

（1）在关键函数优化方面。伏羲在调度资源时支持多个维度，如内存、CPU、网络、磁盘等，所有的资源和请求都用一个多维的键-值对表示，例如{Mem：10，CPU：50，net：40，disk：60}。因此，可以将判断一个空闲资源能否满足一个资源请求的问题简单地抽象成多维向量的比较问题，例如 R：[r1，r2，r3，r4]＞Q：[q1，q2，q3，q4]，其中1、2、3、4等数字表示各个维度，当且仅当 R 的各个维度均大于 Q 时才判断 R＞Q 比较次数决定了这个操作的时间复杂度。在最好的情况下只需比较一次即可得出结果，如判断[1，10，10，10]大于[2，1，1，1]失败；最差需要 D 次（D 为维度数），如判断[10，10，10，1]大于[1，1，1，10]需比较4次。在资源调度高频发生时必须对这里的比较进行优化。通过Profiling分析了系统运行时的资源空闲与请求情况，在资源充足时通常值最大的维度最难满足，因此在资源调度场景采用基于主键的优化算法：将每个资源请求的最大值所在维度定义为该向量的主键，当有空闲资源时首先比较主键维度是否满足请求，如果在主键上满足再比较其他维度。此外，对一个节点上排队等待所有请求的主键值再求一个最小值，空闲资源如果小于该最小值则无须再比较其他请求。通过主键算法大大减少了资源调度时向量的比较次数，伏羲Master一次调度时间优化到了几个毫秒。

（2）在通信性能优化方面。消息从到达伏羲Master进程到最终被处理返回的总时间主要包括在队列中等待的时间和实际处理的时间，所以造成系统高延迟的两个原因是消息处理本身的OPS下降、消息堆积在待处理队列中未被及时处理。在定位到消息堆积的问题后对消息通信策略进行流控，算法简单、有效：发送端检查上次询问的请求结果已经返回，表明目前伏羲Master请求处理较为顺畅，则间隔一个较短的时间后进行下一次询问。反之，如果上次询问的请求超时，说明伏羲Master较忙（例如有任务释放大批资源待处理等），发送端则等待较长时间后再发送请求。通过这种自适应流控的通信策略调整，伏羲Master的消息堆积问题得到了有效解决。

11.6　习题

1. 集群资源统一管理系统需要支持多种计算框架，介绍系统应具备的特点。
2. 简要介绍Slot作业的分类。
3. 相比于“一个计算框架一个集群的模式”，简述共享集群模式的3个优点。
4. 简述分布式计算领域的资源调度策略。
5. 简述YARN的工作机制。
6. 简述阿里云伏羲资源调度架构。

第三部分

大数据分析与应用

第 12 章

数据分析

12.1 数据操作与绘图

R 语言作为一款分析统计软件，擅长对数据进行分析处理，下面介绍 R 的数据结构以及数据的输入方法。

12.1.1 数据结构

R 的运行需要借助于一些对象，一方面是借助于对象的内容和名称，另一方面是借助于对象的数据类型，即属性。对象的属性对于作用于一个对象的函数的表现非常重要，正是这个属性为对象提供了所需要的信息。长度和类型是所有对象都具备的两个内在属性，其中长度指的是对象中元素的数目；类型指的是对象中元素的基本类型，包括字符型、数值型、复数型和逻辑型 4 种。除此之外还有其他不能用来表示数据的类型，如函数或表达式，其中对象分别通过 length 和 mode 函数得到工作长度和类型。例如执行下面的命令并观察相应的输出，如图 12-1 所示。

不管数据类型是哪一种，总是用 NA 来表示缺失数据。可用指数形式来表示很大的数值；R 可以表示无穷的数值，如用 Inf 和-Inf 表示±∞，或者用 NaN 表示不是数字的值。当输入对象为字符型的值时需在值的两端加上双引号。如果在值中出现需要引用双引号的情况时，在引用的双引号的前面需加上反斜杠(\)，这两个合在一起的字符在输出显示或写入磁盘时会按照特殊的方式进行处理。

```
> x <- 1
> mode(x)
  [1] "numeric"
> length(x)
  [1] 1
> A <- "Gomphotherium"; compar <- TRUE; z <- 1i
> mode(A); mode(compar); mode(z)
  [1] "character"
  [1] "logical"
  [1] "complex"
```

图 12-1　简单命令及输出

R 拥有许多用于存储数据的对象类型，包括向量、数组、标量、矩阵、数据框和列表。它们在存储数据的类型、创建方式、结构复杂度以及用于定位和访问的其中个别元素的标记等方面均有所不同。

在描述数据时，对于一个向量，有类型和长度就可以了，但是对于其他对象就没有这么简单了，需要一些由外在的属性给出的额外信息。在这些属性中 dim 表示对象维度，如一个 2 行 2 列的矩阵，它的 dim 是一对数值[2,2]，长度是 4。在 R 语言的数据对象中，向量是一个变量，数据是一个 k 维的数据表，因子是一个分类变量；而矩阵是维数为 2 的数组时间序列，数据包含例如频率和时间等的一些额外的属性，用“ts”来表示，数据框必须是等长的，但可以包含不同的数据类型。需要指出的是，数组或矩阵中的所有元素的类型必须是同一种。

12.1.2　绘图功能

R 的绘图功能非常多样，因为每个绘图函数都有大量的选项，这使得图形的绘制十分灵活、多变，用户可以输入 demo(graphics)或者 demo(persp)来获得详细的信息。绘图函数会将结果直接输出到一个“绘图设备”上，即一个绘图的窗口或一个文件，而不是赋值给一个对象。绘图函数分为低级绘图函数和高级绘图函数两种，低级绘图函数是在现存的图形基础上添加一些元素，而高级绘图函数则是直接创建一个新的图形。

在使用 R 绘图时，首先要打开多个绘图设备，绘图设备可以用适当的函数打开。如果没有打开绘图设备，而绘图函数开始执行，R 将打开一个窗口来展示绘制的图形。操作系统决定了使用哪种绘图设备，在 Windows 系统下称为 windows，在类 UNIX 系统中则使用 x11。因为 x11()可以作为 windows()的别名，所以在 Windows 系统下该命令仍然有效，无论使用的是哪一种操作系统，都可以用 x11()来打开一个绘图窗口。另外，也可以用 pdf()、png()等函数打开一个文件作为绘图设备，最后打开的设备将成为当前的绘图设备。关闭一个设备可以用函数 dev.off()，默认为关闭当前设备，否则表示关闭由

参数指定编号的设备。

若要对图形进行分割，用户可以使用函数 split. screen 分割当前的绘图设备，如图 12-2 所示。

```
> split.screen(c(1, 2))
```

图 12-2　screen 分割

可以将设备划分为两个部分，用 scrren(1)或者 screen(2)进行选择。若要删除最后绘制的图形，可以使用 erase. screen()命令。而使用 split. screen()可以制作复杂的布局，也可以划分设备的一部分。但是只能局限于图形式探索性数据分析之类的问题使用这些函数，并且和其他的函数不兼容，不能用于多个绘图设备。

12.2　初级数据分析

R 中的一些基本统计分析函数可以从 stats 包中获得，包括方差分析、广义线性模型和最小二乘法回归的线性模型、非线性最小二乘法、多元分析、汇总统计、时间序列分析、层次聚类和统计分布。上述统计方法以外的统计方法还可以从其他 R 包中获得，下面从统计分析中非常有用的两个概念(公式和泛型函数)开始介绍。

1. 公式

因为几乎所有函数的符号都一样，所以公式在 R 统计分析里非常重要。

$a:b$	a 和 b 的交互效应
$a+b$	a 和 b 的相加效应
$a*b$	相加和交互效应
$-b$	去掉因子 b 的影响
1	$y\sim 1$ 拟合一个没有因子影响的模型
-1	$y\sim x-1$ 表示通过原点的线性回归
^n	包含到 n 阶的所有交互作用
poly(a,n)	a 的 n 阶多项式

可以看出，在 R 公式里面采用的运算符和表达式里面使用的运算符含义不尽相同。用户也可以在公式中包含一些函数，以便对变量进行一定的转换。

2. 泛型函数

R 中的泛型(genetic)是用来解析结果的，是对特定的类对象有特定行为的函数。R

函数将输入对象作为输入参数，这一点不同于很多其他统计编程语言。泛型函数的优势在于一个函数对所有类的使用格式都是一样的。例如，summary 是最常用的用于解析统计分析结果的 R 函数，它可以显示比较细致的结果。对于作为参数的对象是线性模型（“lm”类）还是方差分析（“aov”类），显示的信息是不一样的。

R 还有一个重要的特性，就是一个包含分析结果的对象常常是一个列表对象，它的类定义决定了它的结果展示方式，即输入参数的对象类型决定一个函数的行为。泛型函数通常是调用自变量所属类的对应函数，其中调用的函数称为方法（method）。

12.2.1 描述性统计分析

在比较多组个体或进行观测时，关注的重点往往是各组的，而不是样本整体的描述性统计信息。同样，在 R 中完成这个任务有很多种方法。我们可使用 Motor Trend 杂志的车辆路试（mtcars）数据集，这里关注的焦点是每加仑汽油行驶的英里数（mpg）、马力（hp）和车重（wt），从获取变速箱类型各水平的描述统计量开始，如图 12-3 和图 12-4 所示。

```
> aggregate(mtcars[vars], by = list(am, mtcars$am), mean)
```

图 12-3 获取统计量命令

```
  am      mpg       hp       wt
1  0 17.14737 160.2632 3.768895
2  1 24.39231 126.8462 2.411000
```

图 12-4 命令输出结果

describe. by()函数不允许指定任意函数，所以它的普适性较低。若存在一个以上的分组变量，可以使用 list(groupvar1, groupvar2,…,groupvarN)来表示它们，示例如图 12-5 所示。但这仅在分组变量交叉后不出现空白单元时有效。

```
> describe.by(mtcars[vars], mtcars$am)
```

图 12-5 describe 函数

数据分析人员对于展示哪些描述性统计变量以及结果采用什么格式都有自己的偏好，用户可以选择最合适的方式，或是创造属于自己的方法。

12.2.2 回归诊断

使用 lm()函数来拟合回归模型，通过 summary()等函数获取模型的参数和相关统

计量。但是，没有任何输出能告诉用户模型是否合适，对模型参数推断的信心依赖于它在多大程度上满足 OLS 模型统计假设。为数据的无规律性或者错误设定了预测变量与相应变量的关系都将使模型产生巨大的偏差，一方面可能得出某个预测变量与相应变量无关的结论，但事实上它们相关；另一方面，情况可能恰好相反。

回归诊断技术向用户提供了评价回归模型适用性的必要工具，它能帮助用户发现并纠正问题。在 R 基础安装中提供了大量检验回归分析中统计假设的方法。最常见的方法就是对 lm()返回的对象使用 plot()函数，可以生成评价模型拟合情况的 4 幅图形。

12.3　交互式数据分析

12.3.1　交互式数据分析的特征

与非交互式数据处理相比，交互式数据处理灵活、直观、便于控制。系统与操作人员以人机对话的方式一问一答——操作人员提出请求，数据以对话的方式输入，系统便提供相应的数据或提示信息，引导操作人员逐步完成所需的操作，直到获得最后处理结果。采用这种方式，存储在系统中的数据文件能够被及时地处理修改，同时处理结果可以被立刻使用。交互式数据处理具备的这些特征能够保证输入的信息得到及时处理，使交互方式继续进行下去。

12.3.2　交互式数据处理的典型应用

在大数据环境下，数据量的急剧膨胀是交互式数据处理系统面临的首要问题。下面以信息处理系统领域和互联网领域作为典型应用场景进行介绍。

(1) 信息处理系统领域。在信息处理系统领域中主要体现了人机之间的交互。传统的交互式数据处理系统主要以关系型数据库管理系统(DBMS)为主，面向两类应用，即联机事务处理(OLTP)和联机分析处理(OLAP)。OLTP 基于关系型数据库管理系统，广泛用于政府、医疗以及对操作序列有严格要求的工业控制领域；OLAP 基于数据仓库系统(data warehouse)，广泛用于数据分析、商业智能(BI)等。其最具代表性的处理是数据钻取，例如在 BI 中可以对数据进行切片和多粒度的聚合，从而通过多维分析技术实现数据的钻取。目前，基于开源体系架构下的数据仓库系统的发展十分迅速，以 Hive、Pig 等为代表的分布式数据仓库能够支持上千台服务器的规模。

(2) 互联网领域。在互联网领域中主要体现了人机之间的交互。随着互联网技术的

发展，传统的简单按需响应的人机互动已不能满足用户的需求，用户之间也需要交互，这种需求诞生了互联网中交互式数据处理的各种平台，如搜索引擎、电子邮件、即时通信工具、社交网络、微博、博客以及电子商务等，用户可以在这些平台上获取或分享各种信息。此外还有各种交互式问答平台，如百度的知道、新浪的爱问等。由此可见，用户与平台之间的交互变得越来越容易、越来越频繁。这些平台中数据类型的多样性使得传统的关系数据库不能满足交互式数据处理的实时性需求。目前，各大平台主要使用 NoSQL 类型的数据库系统来处理交互式的数据，如 HBase 采用多维有序表的列式存储方式，MongoDB 采用 JSON 格式的数据嵌套存储方式。大多数 NoSQL 数据库不提供 Join 等关系数据库的操作模式，以增加数据操作的实时性。

12.3.3 典型的处理系统

交互式数据处理系统的典型代表是 Berkeley 的 Spark 系统和 Google 的 Dremel 系统。

1. Berkeley 的 Spark 系统

Spark 是一个基于内存计算的可扩展的开源集群计算系统。针对 MapReduce 的不足，即大量的网络传输和磁盘 I/O 使得效率较低，Spark 使用内存进行数据计算以便快速地处理查询，实时返回分析结果。Spark 提供比 Hadoop 更高层的 API，同样的算法在 Spark 中的运行速度比 Hadoop 快 10～100 倍。Spark 在技术层面兼容 Hadoop 存储层 API，可访问 HDFS、HBase、SequenceFile 等。Spark-Shell 可以开启交互式 Spark 命令环境，能够提供交互式查询。

Spark 是为集群计算中的特定类型的工作负载设计的，即在并行操作之间重用工作数据集的工作负载。Spark 的计算架构具有以下 3 个特点：

(1) 拥有轻量级的集群计算框架。Spark 将 Scala 应用于它的程序架构，而 Scala 这种多范式的编程语言具有并发性、可扩展性以及支持编程范式的特征，与 Spark 紧密结合能够轻松地操作分布式数据集，并且可以轻易地添加新的语言结构。

(2) 包含大数据领域的数据流计算和交互式计算。Spark 可以与 HDFS 交互取得里面的数据文件，同时 Spark 的迭代、内存计算以及交互式计算为数据挖掘和机器学习提供了很好的框架。

(3) 很好的容错机制。Spark 使用了弹性分布数据集(RDD)，RDD 被表示为 Scala 对象分布在一组节点的只读对象集中，这些集合是弹性的，保证了如果有一部数据集丢失可以对丢失的数据集进行重建。

Spark 高效处理分布数据集的特征使其有很好的应用前景。

2. Google 的 Dremel 系统

Dremel 是 Google 研发的交互式数据分析系统，专注于只读嵌套数据的分析。Dremel 可以组建成规模上千的服务器集群，处理 PB 级数据。传统的 MapReduce 完成一项处理任务最短需要分钟级的时间，而 Dremel 可以将处理时间缩短到秒级，作为 MapReduce 的有力补充，可以通过 MapReduce 将数据导入到 Dremel 中，使用 Dremel 来开发数据分析模型，最后在 MapReduce 中运行数据分析模型。

Dremel 作为大数据的交互式处理系统可以与传统的数据分析或商业智能工具在速度和精度上相媲美。Dremel 系统有以下 5 个特点：

(1) Dremel 是一个大规模系统。在 PB 级数据集上要将任务缩短到秒级，需要大量的并发，而磁盘的顺序读速度在 100MB/s 左右，因此在 1s 内处理 1TB 数据就意味着至少需要有一万个磁盘的并发读。但是机器越多，出问题的概率越大，如此大的集群规模，需要有足够的容错考虑才能够保证整个分析的速度不被集群中的个别慢（坏）节点影响。

(2) Dremel 是对 MapReduce 交互式查询能力不足的有力补充。Dremel 利用 GFS 文件系统作为存储层，经常用它来处理 MapReduce 的结果集或建立分析原型。

(3) Dremel 的数据模型是嵌套(nested)的。Dremel 支持一个嵌套的数据模型，类似于 Json。对于处理大规模数据，不可避免地有大量的 Join 操作，而传统的关系模型显得有心无力，Dremel 却可以很好地处理相关的查询操作。

(4) Dremel 中的数据是用列式存储的。使用列式存储，在进行数据分析的时候可以只扫描所需要的那部分数据，从而减少 CPU 和磁盘的访问量。同时，列式存储是压缩友好的，使用压缩可以综合 CPU 和磁盘，从而发挥最大的效能。

(5) Dremel 结合了 Web 搜索和并行 DBMS 的技术。首先，它借鉴了 Web 搜索中的“查询树”的概念，将一个相对巨大、复杂的查询分割成较小、较简单的查询，分配到并发的大量节点上。其次，与并行 DBMS 类似，Dremel 可以提供一个 SQL-like 的接口，就像 Hive 和 Pig。

12.4 数据仓库与分析

数据仓库和数据库极其相似，都是通过数据库软件基于数据模型来组织、管理数据。但是，数据库更专注于业务交易处理(OLTP)，而数据仓库更专注于数据分析(OLAP)，

因此产生的数据库模型也有很大的差异。数据仓库强调数据分析的效率、复杂查询的速度、数据之间的相关性分析，所以在数据库模型上数据仓库喜欢使用多维模型，从而提高数据分析的效率。从产品实现层面来说，数据仓库倾向于使用列式存储，如 SAP IQ、SAP HANA。IBM、Oracle、Sybase、CA、NCR、Informix、Microsoft 和 SAS 等有实力的公司相继(通过收购或研发的途径)推出了自己的数据仓库解决方案，BO 和 Brio 等专业软件公司也在前端在线分析处理工具市场上占有一席之地。

12.4.1 数据仓库的基本架构

数据仓库用于构建面向分析的集成化数据环境，为企业提供决策支持。其实数据仓库本身并不"生产"任何数据，同时自身也不需要"消费"任何数据，数据来源于外部，并且开放给外部应用，因此数据仓库的基本架构主要包含数据流入与流出的过程，可以分为 3 层——源数据、数据仓库、数据应用，如图 12-6 所示。

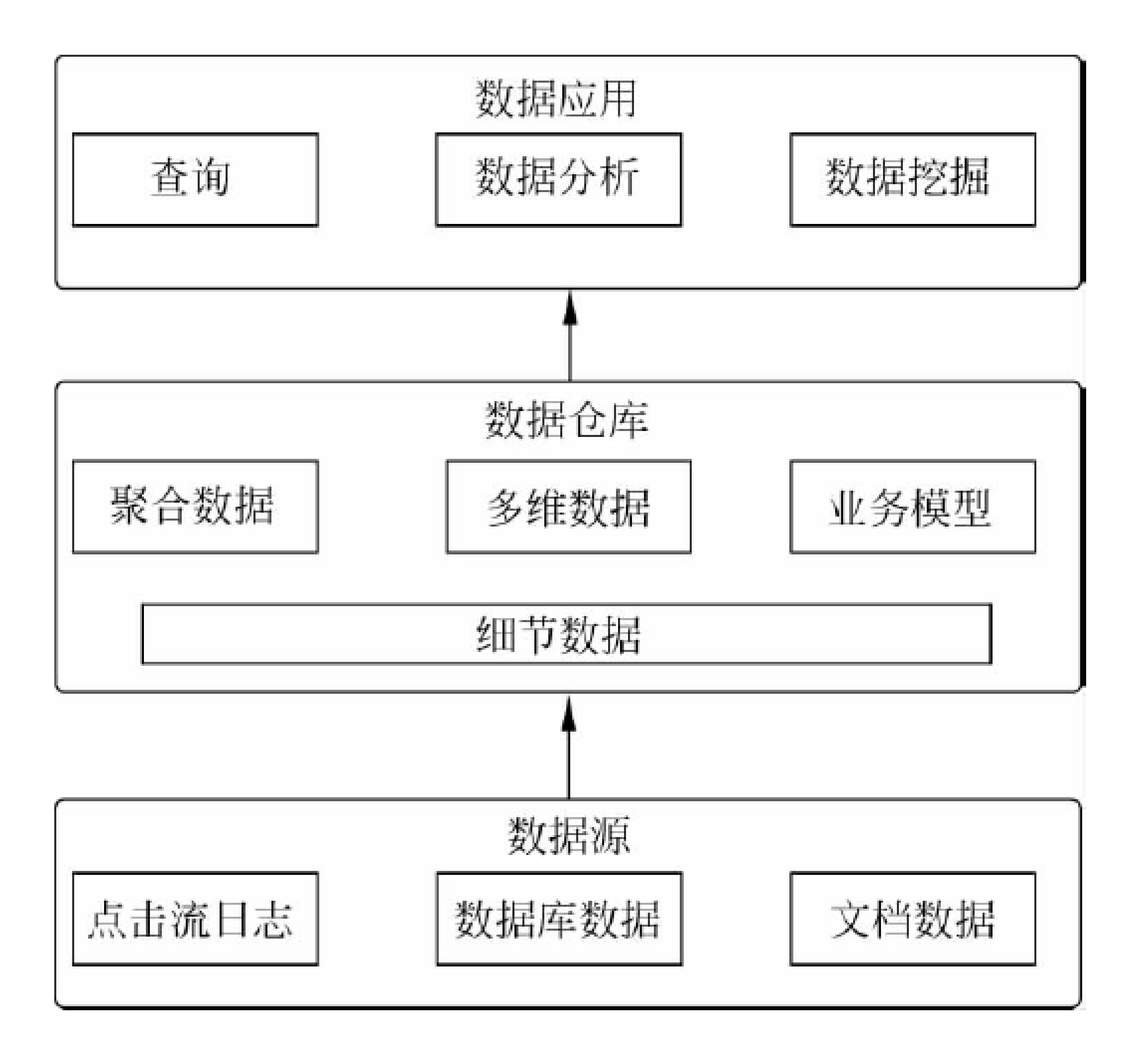


图 12-6 数据仓库的基本架构

12.4.2 数据仓库的实现步骤

1. 确定用户需求

根据终端用户的需求，为数据仓库中存储的数据建立模型。通过数据模型可以得到企业完整而清晰的描述信息，数据模型是面向主题建立的，同时又为多个面向应用的数据源的集成提供了统一的标准。数据仓库的数据模型一般包括企业的各个主题域、主题

域之间的联系、描述主题的码和属性组。

2. 设计和建立数据库

设计和建立数据库是成功创建数据仓库的一个关键步骤,因为涉及的数据来自多种数据源,并且要把它们合并成一个单独的逻辑模型。不像 OLTP 系统那样以高度的正规化形式存储数据,数据仓库以一种非常非正规化的形式存储数据以便提高查询的性能。数据仓库经常使用星形模式和雪花形模式来存储数据,作为 OLAP 工具管理的合计基础,以便尽可能快地响应复杂查询。

星形模式通过使用一个包含主题的事实表和多个包含事实的非正规化描述的维度表来执行典型的决策支持查询。一旦创建了事实表,那么就可以使用 OLAP 工具预先计算常用的访问信息。星形模式是一种关系型数据库结构,在该模式的中间是事实表,周围是次要的表,数据在事实表中维护,维度数据在维度表中维护。每一个维度表通过一个关键字直接与事实表关联。维度是组织数据仓库数据的分类信息,例如时间、地理位置、组织等。维度用于父层和子层这类分层结构。例如,地理位置维度可以包含国家、城市等数据。因此,在该维度表中,纬度由所有的国家、所有的城市组成。

为了支持这种分层结构,在维度表中需要包括每一个成员与更高层次上纬度的关系。维度关键字是用于查询中心事实表数据的唯一标识符。维度关键字就像主键一样,把一个维度表与事实表中的一行链接起来。这种结构很容易构造复杂的查询语句并且支持决策支持系统中向下挖掘式的分析。事实表包含了描述商业特定事件的数据,例如银行业务或者产品销售。事实表还包含了任何数据合计,例如每一个地区每月的销售情况。一般情况下,事实表中的数据是不允许修改的,新数据只是简单地加进去。维度表包含了用于参考存储在事实表中数据的数据,例如产品描述、客户姓名和地址、供应商信息等。把特征信息和特定的事件分开,可以通过减少在事实表中扫描的数据量提高查询性能。维度表不包含与事实表同样多的数据,维度数据可以改变,例如客户的地址或者电话号码改变了。

通过降低需要从磁盘读取数据的数据量,星形模式设计有助于提高查询性能。查询语句通过分析比较小的维度表中的数据来获取维度关键字,以便在中心的事实表中索引,可以降低扫描的数据行。

在转换 OLTP 数据库模式到星形模式时,涉及的步骤如下:

(1) 确定事实表和维度表。

(2) 设计事实表。

(3) 设计维度表。

(4) 实现数据库设计。

3. 提取和加载数据

把操作系统中的数据提取出来然后加载到数据仓库中，其随着复杂性的变化而变化。如果在数据源中的数据和将要出现在数据仓库中的数据直接关联，那么这个进程非常简单。这个进程也可能非常复杂，例如数据源的数据驻留在多个异构系统中，并且在加载数据之前需要大量地转变格式和修改。

1）校验数据

在数据从 OLTP 系统提取之前确保数据完全有效很重要，应该由商业分析人员确定数据源是有效的。对数据的任何变化应该在经营系统中改变，而不是在数据仓库中。校验数据是非常耗时的，通过写存储过程检查数据的域完整性来自动化校验进程。然而，手工校验数据也是必要的。如果发现了无效的数据，应该尽力找到错误发生的原因和更正这些错误。

2）迁移数据

从经营系统中迁移数据一般是在数据复制到数据仓库之前把数据复制到一个中间数据库中。如果数据需要净化，那么把数据复制到中间数据库中是必要的。注意，应该在 OLTP 系统中活动比较低的时候复制数据，否则会降低系统的性能。另外，如果该数据仓库由来自多个相关经营系统中的数据构成，应该确保数据迁移发生在系统同步的时候。如果经营系统不同步，那么数据仓库中的数据可能会产生预想不到的错误。

3）数据净化

数据净化就是使数据达到一致性。在多个经营系统中可能有相同的数据，例如一个名称为 A Company 的公司可能被写成 A Co、A、A Company 等。如果这些名称不一致，那么在查询的时候就会将这个公司作为几个不同的公司处理。如果在数据仓库中的数据生成一致的信息，那么该公司的名称必须完全一致。

4）转换数据

在数据的迁移进程中，经常需要把经营数据转换成一种单独的格式，以适应数据仓库的设计。例如把所有的字母字符转变成大写字母。

12.4.3 分布式数据仓库 Hive

Hive 是建立在 Hadoop 上的数据仓库基础构架，它提供了一系列工具，可以用来进行数据的提取、转化、加载(ETL)，这是一种可以存储、查询和分析 Hadoop 中的大规模数据的机制。Hive 定义了简单的类 SQL 查询语言，称为 HQL，它允许熟悉 SQL 的用户查询数据。同时，该语言也允许熟悉 MapReduce 的开发者开发自定义的 mapper 和

reducer 来处理内建的 mapper 和 reducer 无法完成的复杂分析工作。

Hive 的体系结构如图 12-7 所示，具体如下。

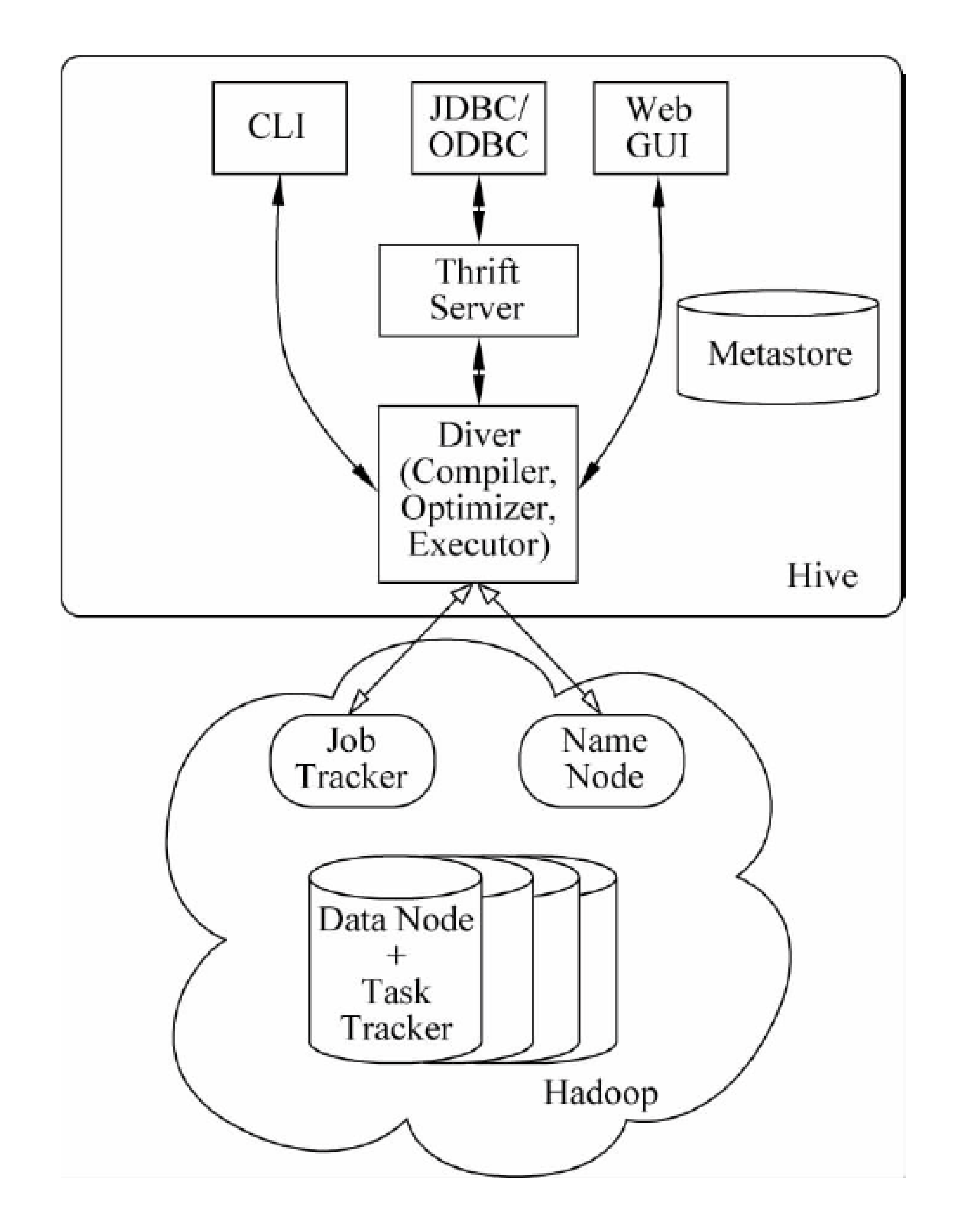


图 12-7　Hive 体系结构图

1. 用户接口

用户接口主要有 3 个，即 CLI、Client 和 WUI，其中最常用的是 CLI，CLI 在启动的时候会同时启动一个 Hive 副本。Client 是 Hive 的客户端，用户连接至 Hive Server。在启动 Client 模式的时候需要指出 Hive Server 所在的节点，并且在该节点启动 Hive Server。WUI 是通过浏览器访问 Hive 的。

2. 元数据存储

Hive 将元数据存储在数据库中，例如 MySQL、Derby。Hive 中的元数据包括表的名字、表的列和分区及其属性、表的属性(是否为外部表等)、表数据所在的目录等。

3. 解释器、编译器和优化器

解释器、编译器和优化器完成 HQL 查询语句从词法分析、语法分析、编译、优化以及查询计划的生成。生成的查询计划存储在 HDFS 中，并在随后由 MapReduce 调用执行。

4. Hadoop

Hive 的数据存储在 HDFS 中,大部分查询由 MapReduce 完成(包含 * 的查询,例如 SELECT * FROM tbl 不会生成 MapReduce 任务)。

12.4.4 数据仓库之 SQL 分析

大数据行业已掌握了收集和记录大量数据的能力,但是根据这些数据进行基本预测和制定决策仍然是一项挑战,需要用简单、易用的数据分析工具来进行数据分析。Apache Hive 就是一种大数据分析工具,通过使用 SQL 方式查找和写入大规模数据到分布式存储系统中,有利于数据仓库数据的查找和更新。

下面以执行 Hive 的 DDL 操作语句和 SQL 操作语句来说明具体的数据仓库操作。

1. DDL(Data Definition Language)操作

数据定义语言用于定义具体的数据表,包括创建、修改、删除表等操作。

首先创建 hive 表:

```
hive> CREATE TABLE pokes(foo INT, bar STRING);
```

该语句创建了一个 hive 表 pokes,该表包括两列,分别是整型的 foo 列和 string 类型的 bar 列。

```
hive> CREATE TABLE invites(foo INT, bar STRING) PARTITIONED BY(ds STRING);
```

该语句创建一个 hive 表 invites,该表包括两列,分别是整型的 foo 列和 string 类型的 bar 列,以及一个分区列,即类型为 string 的 ds 列。其中分区列是一个虚拟的列,并不是数据本身。

```
hive> SHOW TABLES;
```

该语句列出所有的表。

```
hive> SHOW TABLES'.*s';
```

该语句列出所有表的名字以"s"结尾的表。模式匹配遵循 Java 的正则表达式规则。

```
hive>DESCRIBE invites;
```

该语句显示表 invites 中的所有列信息。

```
hive>ALTER TABLE events RENAME TO 3koobecaf;
```

该语句修改表 events 的名字为 3koobecaf。

2. SQL 查询操作

Hive SQL 查询操作语句用于分析具体的数据仓库中的表数据内容。

1）SELECT 语句和过滤操作

```
hive>SELECT a.foo FROM invites a WHERE a.ds = '2008-08-15';
```

该语句表示从表 invites 中输出所有行分区为“2008-08-15”的列信息。查询结果信息输出到终端 console。

2）INSERT 语句操作

```
hive>INSERT OVERWRITE DIRECTORY'/tmp/hdfs_out'SELECT a.* FROM invites a WHERE a.ds = '2008-
08-15';
```

该语句表示从表 invites 中选择分区为“2008-08-15”的所有列信息并同时写入到 HDFS 目录“/tmp/hdfs_out”中。

3）GROUP BY 语句操作

```
hive>INSERT OVERWRITE TABLE events SELECT a.bar;count(*)FROM invites a WHERE a.foo>0
GROUP BY a.bar;
```

4）JOIN 语句操作

```
hive>FROM pokes t1 JOIN invites t2 ON(t1.bar = t2.bar)INSERT OVERWRITE TABLE events SELECT
t1.bar,t1.foo,t2,foo;
```

12.4.5　阿里云 MaxCompute 数据仓库案例

下面以构建一个网站的海量日志为例来说明如何建立一个 MaxCompute 数据仓库。

首先将这些日志收集起来，这里使用 Fluentd 服务（类似的服务还有 Kafka、LogHub、DataX 等），通过 Fluentd 可以轻松地创建任务按时读取各台服务器上的日志文件。用户只需要配置服务器上日志的路径，Fluentd 就能把日志存储到 MaxCompute 的 Table Store 中。

创建的 Table Store 表需要按照 Fluentd 定义的字段写，默认的 4 个字段如下：

(1) content 表示日志内容。

(2) ds 表示天，由 Fluentd 自动生成。

(3) hh 表示小时，由 Fluentd 自动生成。

(4) mm 表示分钟，由 Fluentd 自动生成。

可以看到真正能控制的字段只有 content，其他字段都由 Fluentd 自动生成。

MaxCompute 本身提供了 Fluentd 的所有功能接口，不过调用接口虽然简单，配置环境却很复杂。如果业务需要对数据做深入的分析和挖掘，用户就不得不自己配置环境。

首先需要创建 Table，这和 MySQL 基本一样。代码如下：

```
CREATE TABLE IF NOT EXISTS sale_detail(
shop_name      string,
customer_id    string,
total_price    double)
PARTITION BY(sale_date string, region string);     //设置分区
```

分区一定要考虑到，因为 MaxCompute 的查询最多只能显示 5000 条数据，limit 不支持 offect，所以数据量一大就无法通过开发套件(Data IDE)做在线查询。导出数据到本地也需要使用分区字段，分区越大一次请求能导出的数据就越多，因此，合理地设置分区非常重要。

然后通过 MaxCompute DataHub Service(DHS，通常称为 DataHub 服务)去上传数据。

DataHub 服务提供了 SDK，不过是 Java 的，通过 SDK 可以实现实时上传功能。因为 DataHub 服务接口不用创建 MaxCompute 任务(Task)，所以速度非常快，可以向较高的 QPS(Query Per Second)和较大的吞吐量的服务提供数据存储支持。DataHub 上的数据只能被存储 7 天，之后会被删除，在被删除之前会保存到 MaxCompute 的表中。当然也可通过异步的方式调用接口同步 DataHub 中的数据到 MaxCompute 的表中。

如果服务本身就使用 Java 可以直接使用 SDK，非 Java 服务则需要权衡成本。

在实际项目中使用 MaxCompute 有以下 4 种通用的方案：

(1) 使用 Fluentd 上传数据。其适用于简单的数据存储。

（2）使用 PyODPS 上传数据。其适用于较为复杂的数据存储。

（3）使用 Fluentd 和 PyODPS 上传数据。其适用于需要对数据做大量分析的场景。

（4）使用 DataHub 上传数据。其适用于需要实时同步数据的业务场景。

下面以 PyODPS 为例介绍如何将数据传到 MaxCompute Table 中。

1. 封装 MaxCompute 的连接

这样方便用户随时随地使用 MaxCompute 的强大功能，只需要在代码中调用“odps = OdpsConnect().getIntense()”即可。

```
#/usr/bin/python3
__author__ = 'layne.fyc@gmail.com'
#coding = utf-8
from odps import ODPS
# 测试地址：endpoint = 'http://service-corp.odps.aliyun-inc.com/api'
# 正式地址：endpoint = 'http://service.odps.aliyun-inc.com/api'
debug = True
onlineUrl = 'http://service.odps.aliyun-inc.com/api'
localUrl = 'http://service-corp.odps.aliyun-inc.com/api'
accessId = '填自己的'
accessKey = '填自己的'

class OdpsConnect:
    model = object
    def __init__(self):
        self.model = ODPS(accessId,accessKey,project = '项目名', endpoint = (localUrl if
debug else onlineUrl))
    def getIntense(self):
        return self.model
    def exe(self,sql):
        return self.model.execute_sql(sql)
```

2. 创建 MaxCompute Table

创建 MaxCompute Table，用来存储海量日志。

```
from OdpsConnect import OdpsConnect
from odps.models import Schema, Column, Partition
```

```
odps = OdpsConnect().getIntense()
odps.delete_table('test_amap_analys', if_exists = True)   #表存在时删除

#各种项
columns = [
    Column(name = 'uid', type = 'bigint', comment = 'user id'),
    Column(name = 'ctime', type = 'bigint', comment = 'time stamp'),
    Column(name = 'url', type = 'string', comment = 'url'),
    Column(name = 'param', type = 'string', comment = 'param'),
    Column(name = 'ip', type = 'string', comment = 'ip'),
    Column(name = 'city', type = 'string', comment = 'city')
]
#分区信息
partitions = [
    Partition(name = 'dt', type = 'bigint', comment = 'the partition day')
]
schema = Schema(columns = columns, partitions = partitions)
table = odps.create_table('test_amap_analys', schema, if_not_exists = True)
```

这里有几点需要特别注意：

(1) MaxCompute Table 只支持添加数据，不支持删除与修改数据。如果要删除脏数据，只能先备份整张表到 B，然后删除这张表 A，再新建表 A，最后将表 B 的备份信息处理后重新导入表 A。

(2) 分区信息可以创建很多个，但是在导入、导出、进行某些特殊查询时要全部带上。例如分区字段为“a，b，c，d”，最后导出数据时必须指定“a，b，c，d”的内容，只指定“a，b”或者“a. c”都是不行的。所以用户设置分区字段时也要慎重，要尽量设置少一点，这里通过数据量来设置，建议每个分区存储 20 000 条左右的数据。

3. 通过 Tunnel 上传数据到 MaxCompute

描述：通过程序在服务器上存放了大量日志文件，文件名为“log/20160605. log”，每天通过日期定时生成。每条日志的格式如下：

```
name:amap|ip:19.19.19.19|uuid:110112|param:sdfsdf = 123&fsdf = 123|url:get - user - info
|time:123123123
```

使用“|”和“:”号分隔，下面需要将所有的日志数据存储到上一步创建的 test_amap_analys 表中。

```
#/usr/bin/python3
__author__ = 'layne.xfl@alibaba-inc.com'
#coding=utf-8
import datetime
import urllib.parse
#自己封装的 IP 库
import db.IP
from OdpsConnect import OdpsConnect
from odps.models import Schema, Column, Partition
from odps.tunnel import TableTunnel

odps = OdpsConnect().getIntense()
t = odps.get_table('test_amap_analys')
records = []
tunnel = TableTunnel(odps)
#日志文件名格式为 log/20160605.log
#这里默认上传前 5 天的数据
sz = 5
while sz > 0 :
    #get Yestoday
    last_time = (datetime.datetime.now() + datetime.timedelta(days=-sz)).strftime
('%Y%m%d')
    #分区不存在的时候需要创建,否则会报错
    t.create_partition('dt='+last_time, if_not_exists=True)
    #创建连接会话
    upload_session = tunnel.create_upload_session(t.name, partition_spec='dt='+
last_time)
    #通过日期规则构造的文件名
    file = 'log/%s.log' % last_time
    with upload_session.open_record_writer(0) as writer:
        for line in open(file,'r',encoding='utf8'):
            arr = {}
            #这里的日志使用"|"和":"号分隔
            #例如:"name:amap|ip:19.19.19.19|uuid:110112|param:sdfsdf=123&fsdf=
123|url:get-user-info|time:123123123"
            for tm in raw.split('|'):
                pstm = tm.split(':')
                if(len(pstm)==2):
                    arr[pstm[0]] = pstm[1]
```

```
            #Nginx 中的 ip 参数可能被伪造,需要做过滤
            ip = arr.get('ip','')
            if(len(ip)>15):
                iparr = ip.split(',')
                ip = iparr[-1].strip(" ")
            city = ''
            if ip != '':
                #通过 ip 获取城市信息
                city = db.IP.find(ip)
            writer.write(t.new_record(
                [
                    arr.get('uid',0),
                    arr.get('time',0),
                    arr.get('url',''),
                    urllib.parse.unquote(arr.get('param','')).replace("\\","*").
replace('"',"*"),  #做 URL_DECODE
                    ip,
                    city,
                    last_time
                ]))
    upload_session.commit([0])
    sz = sz - 1;
```

这里需要注意以下两点:

(1) 使用 MaxCompute 要以数据为重,分区先行。存储数据、下载数据都要先设置好分区再操作数据。

(2) Nginx 中获取的 IP 参数可能被伪造,不能直接使用。

12.5 习题

1. 简述数据分析过程。
2. 简述数据仓库定义、数据仓库架构以及实现步骤。
3. 简述 Hive 的基本功能。
4. 使用阿里云数据仓库 MaxCompute 构建一个日志分析数据仓库,分析日志数据。

第 13 章

数据挖掘与机器学习技术

13.1　相关理论基础知识

13.1.1　数据挖掘与机器学习简介

数据挖掘包括信息收集、数据集成、数据规约、数据清理、数据变换、数据挖掘过程、模式评估、知识表示 8 个步骤。数据挖掘过程是一个反复循环的过程，每一个步骤如果没有达到预期目标，都需要回到前面的步骤重新调整并执行。不是每个数据挖掘工作都需要包括这里列出的每一步，例如在某个工作中不存在多个数据源，那么数据集成的步骤就可以省略。数据规约、数据清理、数据变换合称为数据预处理。在数据挖掘中，至少60%的费用可能花在信息收集阶段，而至少 60%以上的精力和时间花在数据预处理上。

机器学习的目的则是从数据中自动习得模型，并使用习得的模型对未知数据进行预测。机器学习的任务是从数据中学习决策函数 $f:x\rightarrow y$，这个决策函数将输入变量 x 映射到输出空间的输出变量 y 上，即根据输入产生预测。

机器学习包括监督学习、非监督学习、半监督学习以及强化学习等，其中最常见的两类任务是监督学习与非监督学习。

监督学习的任务是利用训练数据学习决策函数 f，并将其应用于测试数据上进行推理和预测。一般而言，训练数据由标注好的输入和输出数据对 $(x,y)\in X\times Y$ 构成，训练数据集通常表示为：

$$\tau=\{(x_1,y_1),(x_2,y_2),\cdots,(x_n,y_n)\}$$

在监督学习的过程中，学习系统利用事先标注好的特定训练数据集，通过学习得到某个模型，这个模型可以表示为条件概率分布或者决策函数，条件概率分布或者决策函数描述输入和输出之间随机变量的映射关系。监督学习的最终目标是使从训练数据中习得的模型能够在测试数据上获得准确的预测能力。形式化地讲，对于给定的输入 x，决策函数 f 产生的预测值 $f(x)$ 与真实值可能一致，也可能不一致，一般使用损失函数(Loss Function)来度量预测的错误程度。损失函数是预测值 $f(x)$ 与真实值 y 的非负实值函数，记作 $\mathcal{L}(y,f(x))$。

常见的损失函数有以下 4 个。

(1) 0-1 损失函数

$$\mathcal{L}(y,f(x)) = \begin{cases} 1, & y \neq f(x) \\ 0, & y = f(x) \end{cases}$$

(2) 平方损失函数

$$\mathcal{L}(y,f(x)) = (y - f(x))^2$$

(3) 绝对损失函数

$$\mathcal{L}(y,f(x)) = | y - f(x) |$$

(4) 对数损失函数

$$\mathcal{L}(y,f(x)) = - \log P(x \mid y)$$

很明显，损失函数值越小，习得的模型越好。当给定训练数据集合时，模型关于训练数据的平均损失成为经验风险，一般记为：

$$R_{emp}(f)^n = \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}(y_i, f(x_i))$$

经验风险最小化策略认为：经验风险最小的模型就是最优模型。

13.1.2　关联分析

关联分析是一种简单、实用的分析技术，就是发现存在于大量数据集中的关联性或相关性，从而描述一个事物中某些属性同时出现的规律和模式。

关联分析是从大量数据中发现项集之间有趣的关联和相关联系。关联分析的一个典型例子是购物篮分析。该过程通过发现顾客放入其购物篮中的不同商品之间的联系分析顾客的购买习惯，通过分析了解哪些商品频繁地被顾客同时购买，这种关联的发现可以帮助零售商制定营销策略。其他的应用还包括价目表设计、商品促销、商品的摆放和基于购买模式的顾客划分。

1. 基本概念

(1) 项集。在关联分析中,包含 0 个或者多个的项的集合称为项集。如果一个项集包含 k 个项,那么就称为 k 项集。例如,{牛奶,咖啡}称为 2 项集。

(2) 支持度。支持度用来确定给定数据集的频繁程度,即给定数据集在所有的数据集中出现的频率,例如 s(X —> Y) = P(X,Y)/N。

(3) 置信度。置信度则是用来确定 Y 在包含 X 的事务中出现的频繁程度,即 c(X—>Y)=P(X,Y)/P(X)。

2. 关联分析算法的基本原理

支持度是一个重要的度量,如果支持度很低,代表这个规则只是偶然出现,基本没有意义。因此,支持度通常用来删除那些无意义的规则。置信度则是通过规则进行推理,具有可靠性。用 c(X—> Y)来说,只有置信度越高,Y 出现在包含 X 的事务中的概率才越大,否则这个规则也没有意义。

在做关联规则发现的时候通常会设定支持度和置信度阈值(minsup 和 minconf),而关联规则发现则是发现那些支持度大于等于 minsup 并且置信度大于 minconf 的所有规则。所以,提高关联分析算法效率的最简单的办法是提高支持度和置信度的阈值。

3. 关联分析基本算法

- 找到满足最小支持度阈值的所有项集,通常称为频繁项集(例如频繁 2 项集、频繁 3 项集)。
- 从频繁项集中找到满足最小置信度的所有规则。
- 评估关联分析算法。

1) Apriori 算法

Apriori 算法是挖掘产生布尔关联规则所需频繁项集的基本算法,也是最著名的关联规则挖掘算法之一。Apriori 算法就是根据有关频繁项集特性的先验知识而命名的。它使用一种被称为逐层搜索的迭代方法,k 项集用于探索(k+1)项集。首先找出频繁 1 项集的集合,记做 L_1,L_1用于找出频繁 2 项集的集合 L_2,再用于找出 L_3,如此下去,直到不能找到频繁 k 项集。找每个 L_k需要扫描一次数据库。

为提高按层次搜索并产生相应频繁项集的处理效率,Apriori 算法利用了一个重要性质,并应用 Apriori 性质来帮助有效地缩小频繁项集的搜索空间。

Apriori 性质:一个频繁项集的任一子集也应该是频繁项集。

证明根据定义,若一个项集 I 不满足最小支持度阈值 min_sup,则 I 不是频繁的,即

P(I)< min_sup。若增加一个项 A 到项集 I 中,则结果新项集(I∪A)也不是频繁的,在整个事务数据库中所出现的次数也不可能多于原项集 I 出现的次数,因此 P(I∪A)< min_sup,即(I∪A)也不是频繁的。这样就可以根据逆反公理很容易地确定 Apriori 性质成立。

Apriori 算法的优化有以下几种方法。

(1) 基于划分的方法。该算法先把数据库从逻辑上分成几个互不相交的块,每次单独考虑一个分块并对它生成所有的频繁项集,然后把产生的频繁项集合并,用来生成所有可能的频繁项集,最后计算这些项集的支持度。这里分块的大小选择要使每个分块可以被放入主存,每个阶段只需要扫描一次。而算法的正确性是由每一个可能的频繁项集至少在某一个分块中是频繁项集来保证的。

上面所讨论的算法是可以高度并行的,可以把每一分块分别分配给某一个处理器生成频繁项集。在产生频繁项集的每一个循环结束后,处理器之间进行通信来产生全局的候选是 1 项集。通常这里的通信过程是算法执行时间的主要瓶颈。另一方面,每个独立的处理器生成频繁项集的时间也是一个瓶颈。其他的方法还有在多处理器之间共享一个杂凑树来产生频繁项集,更多关于生成频繁项集的并行化方法可以在其中找到。

(2) 基于 Hash 的方法。Park 等人提出了一个高效地产生频繁项集的、基于杂凑(Hash)的算法。通过实验可以发现,寻找频繁项集的主要计算是在生成频繁 k 项集 L_k 上,Park 等人就是利用这个性质引入杂凑技术来改进产生频繁 2 项集的方法。

(3) 基于采样的方法。基于前一遍扫描得到的信息,对它详细地做组合分析,可以得到一个改进的算法。其基本思想是:先使用从数据库中抽取出来的采样得到一些在整个数据库中可能成立的规则,然后对数据库的剩余部分验证这个结果。这个算法相当简单,并且显著地减少了 I/O 代价,但是一个很大的缺点就是产生的结果不精确,即存在所谓的数据扭曲(Dataskew)。分布在同一页面上的数据经常是高度相关的,不能表示整个数据库中模式的分布,由此导致的是采样 5% 的交易数据所花费的代价与扫描一遍整个数据库相近。

(4) 减少交易个数。减少用于未来扫描的事务集的大小,基本原理就是当一个事务不包含长度为 k 的大项集时,必然不包含长度为 $k+1$ 的大项集。从而可以将这些事务删除,在下一遍扫描中就可以减少要进行扫描的事务集的个数。这就是 AprioriTid 的基本思想。

2) FP-growth 算法

由于 Apriori 算法的固有缺陷,即使进行了优化,其效率仍然不能令人满意。2000 年,Han Jiawei 等人提出了基于频繁模式树(Frequent Pattern Tree,FP-tree)发现频繁模式的算法 FP-growth。在 FP-growth 算法中,通过两次扫描事务数据库把每个事务所包

含的频繁项目按其支持度降序压缩存储到 FP-tree 中。在以后发现频繁模式的过程中，不需要再扫描事务数据库，而仅在 FP-tree 中进行查找即可，并通过递归调用 FP-growth 的方法来直接产生频繁模式，因此在整个发现过程中也不需要产生候选模式。该算法克服了 Apriori 算法中存在的问题，在执行效率上也明显优于 Apriori 算法。

13.1.3　分类与回归

1. 分类问题

分类是数据挖掘的一种非常重要的方法。分类的概念是在已有数据的基础上学会一个分类函数或构造出一个分类模型（即通常所说的分类器）。该函数或模型能够把数据库中的每一条数据样本映射到给定类别中的某一个，从而可以应用于数据预测。分类器是数据挖掘中对样本进行分类的方法的统称，包含逻辑回归、决策树、朴素贝叶斯、神经网络等算法。

分类的定义：给定一个数据集 $D=\{t_1,t_2,\cdots,t_n\}$ 和一组类 $C=\{C_1,C_2,\cdots,C_n\}$，分类问题就是去确定一个映射 $f: D->C$，每个元组 t_i 被分配到一个类中。类 C_j 包含映射到该类中的所有数据元组，即 $C_j=\{t_i \mid f(t_i)=C_j, 1<=i<=n$，且 $t_i \in D\}$。

分类的过程描述如下。

（1）选定样本（包含正样本和负样本），将所有样本分成训练样本和测试样本两部分。

（2）在训练样本上执行分类器算法，生成分类模型。

（3）在测试样本上执行分类模型，生成预测结果。

（4）根据预测结果计算必要的评估指标，评估分类模型的性能。

下面介绍几种基本的分类器。

1）逻辑回归分类器

二项逻辑回归模型是一种分类模型，可由条件概率分布 $P(y|X)$ 来表示，其中，随机变量 X 的取值范围为实数，随机变量 y 的取值为 -1 或者 1。一般采用以下公式来表示：

$$P(y=1 \mid X)=\frac{\exp(W*X+b)}{1+\exp(W*X+b)}$$

$$P(y=-1 \mid X)=\frac{1}{1+\exp(W*X+b)}$$

其中，X 是输入变量，y 是输出，代表不同的类标号，W 是特征权重向量。

对于 CTR 预估来说，可以采用以上的逻辑回归方法，在计算广告的应用环境下，$y=1$ 代表用户会点击广告，$y=-1$ 代表用户不会点击广告，变量 X 代表<查询，广告>数据对，由数据对响应的特征向量构成，其特征可以是查询独有的、广告独有的或者查询和广告两者匹配的特征，W 是这些特征的权重向量，表明对应特征的重要性，需要使用的训练数据

经过训练获得,特征参数规模可以达到 10 亿量级。当特征权重确定后,输入<查询,广告>数据对,计算出的 $P(y=1|X)$就代表预估出的 CTR。很明显,这是一个 sigmod 函数,其值落在 0 到 1 之间。

所以,问题即转换为如何求解特征权重。可以使用<查询,广告>的历史点击数据作为训练集合,利用极大似然法来估计模型参数:

$$\prod_{j=1}^{M} P(y_i \mid X_j, W)$$

其中(X_j, W)为 M 个训练数据中的第 j 个训练实例,分为正样本和负样本,通过对以上公式取负 log,并为避免过拟合而加入L_1正则化项,则求解特征权重问题转换为用以下公式求最小值:

$$\mathrm{Min}_\beta - \sum_{j=1}^{M} \log(P\, y_i \mid X_j, W) + \lambda \| W \|$$

于是,问题就变成了以对数似然函数为目标的无约束最优化问题,一般采用梯度下降算法、牛顿拟合法或拟牛顿法来解决这种参数优化的问题。不论采取上述哪种参数优化方法都遵循相似的基本流程:首先赋予初始的权重向量W_0以 0 附近的随机值,然后反复迭代,在每次迭代过程中都根据当前权重向量 W 计算目标函数的最快下降方向,并更新权重向量,直到目标函数稳定到极值点,将此时的权重向量 W 作为目标函数的最优解。

不同优化算法的主要区别在于目标函数下降方向D_1的计算方式不同,D_1往往是通过对目标函数权重向量 W 的当前取值求导数或者二阶导数来确定的。

2) 决策树分类器

决策树分类器提供一个属性集合,决策树通过在属性集的基础上做出一系列的决策将数据分类。这个过程类似于通过一个植物的特征来辨认植物。可以应用这样的分类器来判定某人的信用程度,例如,一个决策树可能会断定“一个有家、拥有一辆价值在 1.5 万~2.3 万美元的轿车、有两个孩子的人”拥有良好的信用。决策树生成器从一个“训练集”中生成决策树。SGI 公司的数据挖掘工具 MineSet 所提供的可视化工具使用树图来显示决策树分类器的结构,在该图中,每一个决策用树的一个节点来表示。图形化的表示方法可以帮助用户理解分类算法,提供对数据的有价值的观察视角。生成的分类器可用于对数据的分类。

3) 选择树分类器

选择树分类器使用与决策树分类器相似的技术对数据进行分类。与决策树不同的是,选择树中包含特殊的选择节点,选择节点有多个分支。例如,在一棵用于区分汽车产地的选择树中的一个选择节点可以选择马力、汽缸数目或汽车重量等作为信息属性。在决策树中,一个节点一次最多可以选取一个属性作为考虑对象。在选择树中进行分类时

可以综合考虑多种情况。选择树通常比决策树更准确，但是也大得多。选择树生成器使用与决策树生成器生成决策树同样的算法，从训练集中生成选择树。MineSet 的可视化工具使用树图来显示选择树。树图可以帮助用户理解分类器，发现哪个属性在决定标签属性值时更重要，同样可以用于对数据进行分类。

4）证据分类器

证据分类器通过检查在给定一个属性的基础上某个特定的结果发生的可能性来对数据进行分类。例如，它可能做出判断，一个拥有一辆价值在 1.5～2.3 万美元的轿车的人有 70％的可能是信用良好的，有 30％的可能是信用很差。分类器在一个简单的概率模型的基础上使用最大的概率值来对数据进行分类预测。与决策树分类器类似，生成器从训练集中生成证据分类器。MineSet 的可视化工具使用证据图来显示分类器，证据图由一系列描述不同的概率值的饼图组成。证据图可以帮助用户理解分类算法，提供对数据的深入洞察，帮助用户回答像“如果…怎么样”一类的问题，同样可以用于对数据进行分类。

2. 回归问题

回归分析(regression analysis)是确定两种或两种以上变数间相互依赖的定量关系的一种统计分析方法，其运用十分广泛。回归分析按照涉及的自变量的多少可以分为一元回归分析和多元回归分析；按照自变量和因变量之间的关系类型可以分为线性回归分析和非线性回归分析。如果在回归分析中只包括一个自变量和一个因变量，且两者的关系可用一条直线近似表示，这种回归分析称为一元线性回归分析。如果回归分析中包括两个或两个以上的自变量，且因变量和自变量之间是线性关系，则称为多元线性回归分析。通过这种方法可以确定许多领域中各个因素(数据)之间的关系，从而可以通过其预测和分析数据。例如司机的鲁莽驾驶与道路交通事故数量之间的关系，最好的研究方法就是回归。

回归的原理是：建立因变数 Y(或称依变数、反应变数)与自变数 X(或称独变数、解释变数)之间关系的模型。简单线性回归使用一个自变量 X，复回归使用超过一个自变量($X_1, X_2, \cdots, X_i$)。

回归模型主要包括以下变量：

(1) 未知参数 β，可以是标量或向量。

(2) 自变量 X。

(3) 因变量 Y。

回归模型将 Y 和一个关于 X 和 β 的函数关联起来。

$$Y \approx f(X, \beta)$$

13.1.4 聚类分析

聚类分析(cluster analysis)是一组将研究对象分为相对同质的群组(clusters)的统计分析技术。聚类分析区别于分类分析(classification analysis),后者是有监督的学习,而聚类分析是无监督的学习。

1. 聚类方法的分类

(1) 层次聚类。包括合并法、分解法、树状图。

(2) 非层次聚类。包括划分聚类、谱聚类。

K-means(非层次聚类)的执行过程如下。

(1) 初始化。选择(或人为指定)某些记录作为凝聚点。

(2) 循环。按就近原则将其余记录向凝聚点凝集,计算出各个初始分类的中心位置(均值),用计算出的中心位置重新进行聚类,如此反复循环,直到凝聚点位置收敛为止。

其方法特点如下。

(1) 通常要求已知类别数。

(2) 可人为指定初始位置。

(3) 节省运算时间。

(4) 只能使用连续性变量。

2. 聚类算法的分类

(1) 划分方法(PArtitioning Method,PAM)。首先创建 k 个划分,k 为要创建的划分个数;然后利用循环定位技术通过将对象从一个划分移到另一个划分来帮助改善划分质量。

(2) 层次方法(hierarchical method)。创建一个层次以分解给定的数据集。该方法可以分为自上而下(分解)和自下而上(合并)两种操作方式。为弥补分解与合并的不足,层次合并经常要与其他聚类方法相结合,如循环定位。

(3) 基于密度的方法。根据密度完成对象的聚类,它根据对象周围的密度(如DBSCAN)不断增长聚类。

(4) 基于网格的方法。首先将对象空间划分为有限个单元以构成网格结构,然后利用网格结构完成聚类。

(5) 基于模型的方法。它假设每个聚类的模型并发现适合相应模型的数据。

13.1.5 离群点检测

定义：在样本空间中与其他样本点的一般行为或特征不一致的点称为离群点。

1. 离群点产生的原因

(1) 计算的误差或者操作的错误所致。例如某人的年龄为－10岁，这就是明显由误操作所导致的离群点。

(2) 数据本身的可变性或弹性所致。例如一个公司中CEO的工资肯定明显高于其他普通员工的工资，于是CEO变成由于数据本身可变性所导致的离群点。

2. 离群点检测方法

1) 基于统计分布的离群点检测

统计学方法是基于模型的方法，即为数据创建一个模型，并且根据对象拟合模型的情况来评估它们。大部分用于离群点检测的统计学方法都是构建一个概率分布模型，并考虑对象有多大可能符合该模型。

离群点的概率定义是：离群点是一个对象关于数据的概率分布模型，它具有低概率。这种情况的前提是用户必须知道数据集服从什么分布，如果估计错误就造成了重尾分布。

异常检测的混合模型方法是：对于异常检测，数据用两个分布的混合模型建模，一个分布为普通数据，另一个为离群点。

聚类和异常检测目标都是估计分布的参数，以最大化数据的总似然(概率)。在聚类时，使用EM算法估计每个概率分布的参数。然而这里提供的异常检测技术使用一种更简单的方法，初始时将所有对象放入普通对象集，而异常对象集为空，然后用一个迭代过程将对象从普通集转移到异常集，只要该转移能提高数据的总似然(其实等价于把在正常对象的分布下具有低概率的对象分类为离群点)。假设异常对象属于均匀分布，异常对象由这样一些对象组成，这些对象在均匀分布下比在正常分布下具有显著更高的概率。

2) 基于距离的离群点检测

基于距离的离群点检测指的是，如果样本空间D中至少有 N 个样本点与对象O的距离大于dmin，那么称对象O是以{至少 N 个样本点}和dmin为参数的基于距离的离群点。

其实可以证明，在大多数情况下，如果对象O是根据基于统计的离群点检测方法发

现的离群点，那么肯定存在对应的 N 和 dmin，于是它也成为基于距离的离群点。

这个方法的缺点是：要求数据分布均匀，当数据分布不均匀时，基于距离的离群点检测将遇到困难。

3）基于密度的局部离群点检测

一个对象如果是局部离群点，那么相对于它的局部领域，它是远离的。

不同于前面的方法，基于密度的局部离群点检测不将离群点看作一种二元性质，即不是简单地用 Yes 或 No 来断定一个点是否为离群点，而是用一个权值来评估它的离群度。

它是局部的，该程度依赖于对象相对于其领域的孤立情况。使用这种方法可以同时检测出全局离群点和局部离群点。

通过基于密度的局部离群点检测能在样本空间数据分布不均匀的情况下准确地发现离群点。

4）基于偏差的离群点检测

基于偏差的离群点检测通过检查一组对象的主要特征来识别离群点，具有“偏差”这种特征的点就认为是离群点。

通常有两种技术：顺序异常技术和 OLAP 数据立方体技术。

13.1.6 复杂数据类型的挖掘

复杂数据类型的挖掘包括复杂对象、空间数据、多媒体数据、时间序列数据、文本数据和 Web 数据。

复杂结构化数据的存取方法在对象关系和面向对象数据库系统中已有研究。在这些系统中，大量的复杂数据对象组织为类，类又按类/子类的层次加以组织。类中的每个对象具有：一个对象标识；一组属性，它们可以具有复杂的数据结构，如集合（set）值或列表（list）值数据、类复合层次（class composition hierarchies）、多媒体数据等；一组方法，用于说明与对象类相关的计算程序或规则。

对象关系和面向对象数据库的主要特征就是对复杂结构数据（如集合值和列表值数据，以及具有嵌套结构的数据）的存储、访问和建模。

下面介绍数据的概化。

1. 集合值属性概化

将集合中的每一个值概化为其对应的更高级别的概念或者导出集合的一般特征，如集合中元素的个数、集合中类型或值的区间分布或数字数据的加权平均。

集合值属性可以概化为集合值属性或单值属性；若单值属性形成一个格(lattice)或“层次”，或概化有不同的概化路径，则它可以概化为一个集合值属性；进一步说，在概化集合值属性上的概化应遵循集合中每一个值的概化路径。

2. 列表值属性概化

与集合值属性类似，在概化中要保持元素的次序。列表中的每一个值可以概化为其对应的高级别概念，或者把一个列表概化为一般特征，如列表长度、列表元素类型、值区间、数字值的加权平均，或删除列表中不重要的元素。一个列表可以概化为列表、集合或单一值。

13.2 应用实践

13.2.1 广告点击率预测

竞价广告是商业搜索引擎公司的主要收入来源，广告商提供创意并对关键词竞价，当搜索引擎用户发出搜索关键词时触发购买了该关键词的厂商广告，对于多个被触发的广告存在一个广告创意排名的问题，即需要确定应该将哪些广告展示出来以及其展示顺序。

CTR(Click Through Rate)代表广告的点击率，一般用广告点击量和展示次数的比率来计算，其表征广告和用户需求的匹配程度，CTR越高，广告排名越高。至于如何估算CTR，一种直观的想法是根据广告的历史点击率信息来进行估算。对于某个查询词，假设广告A在1000次展示中被点击了10次，那么可以估算其CTR为0.0.1。这种思路看似简单，但是并不可行，主要原因是使用历史数据量不足以估算出可信数值。因此必须采用一定的技术手段来合理地估算广告的CTR，一般做法是利用逻辑回归等机器学习模型来根据查询和广告的特征对CTR进行预估。

阿里妈妈目前负责阿里的广告业务，他们希望提高广告CTR预估的准确性，但面临两大挑战：一是数据规模庞大，涉及百亿级别的记录；二是数据每天都在更新，对模型性能要求更高。目前基于ODPS机器学习框架，通过逻辑回归算法实现模型训练，然后把训练结果输出给线上服务的广告投放引擎。

13.2.2 并行随机梯度下降

并行随机梯度下降是Yahoo!提出的一种简单的基于数据并行的随机梯度下降

算法。

假设以c_i代表训练数据，其中 $1\leqslant i\leqslant m$，以 η 代表固定的学习率，T 代表设定的阈值，W_i代表第 i 轮迭代时的特征权重向量，假设有 k 台可并发运行的随机梯度下降（SGD）机器，则其执行逻辑为：首先总控程序将所有的续联数据分发到 k 台服务器，每台服务器获得完整的训练数据，然后每台服务器可并行执行单机 SGD 任务，单机 SGD 任务顺序地从所有的训练数据中随机抽样，并根据其梯度的反方向作为目标函数下降方向，每次以 η 步长更新特征权重向量，如此反复执行 T 次，每台服务器的单机 SGD 任务可以得到 yield 模型参数，之后总控程序从每台服务器收集到各自的模型，通过取均值的方式获得最终的模型参数。

尽管这个方法看上去非常简单，但是论文作者证明了这个算法的可收敛性。

13.2.3 自然语言处理：文档相似性的计算

计算文档集合内任意两个文档的相似性在自然语言处理中是非常常见的应用场景。典型的计算任意两个文档之间的相似性的计算复杂度是 $O(n^2)$，如果文档集合较小，是可以单机处理的，但是如果文档集合规模较大，如百万级别，那么只能考虑通过多机并行来进行计算。这是一种典型的非迭代式批处理任务，所以很适合使用 MapReduce 来解决。

下面介绍如何计算两个文档的文本相似性。在文本处理任务中，一般将一个文档 d 在内部以特征向量W^d来表示，每个单词 t 对应特征向量中的一维，以$W_{t,d}$来表征单词 t 在文档 d 中的权重。通过这种方式，每个文档都表达为特征向量，可以使用两个特征向量的内积来表示两个文档的文本相似性，即如下公式：

$$\mathrm{sim}(d_i,d_j)=\sum_{t\in V}W_{t,i}*W_{t,j}$$

其中，V 是单词集合的大小，即特征向量的维数。从上面的公式可以看出，只有当某个单词同时出现在两个文档中时其对应维度特征的乘积才不为 0。有了计算文档相似性的公式，就可以在此基础上计算文档集合内任意两个文档的文本相似性。

一种简单、粗暴的方式是对任意两个文档根据公式进行计算，但是其实可以对公式进行优化。优化的基本思路是：对于任意两个文档来说，考虑任意单词 t 对于其相似性计算的影响，很明显，如果 t 在任意一个文档中没有出现，则其对两者相似性的贡献为 0，即只有两个文档同时包含单词 t，那么单词 t 才对这两者的相似性有影响。如果从单词 t 的角度出发，能够记录哪些文档包含了单词 t 并形成一个集合，那么对单词 t 来说，其只对集合中文档的两两相似性做出贡献。

如果文档的集合较大，可以考虑使用两个连续的 MR 任务来完成上述计算流程。第一个 MR 任务是建立单词索引的过程，任务的输入是文档 d，Map 阶段输出的 Key 为单

词 t,Value 为包含这个单词的文档 ID 及单词权重;Reduce 阶段将包含单词 t 的文档进行汇总,并将倒排索引写入磁盘。有了单词的倒排索引,则第二个阶段 MR 任务即可计算文档相似性。其输入为第一个阶段 MR 任务输出的单词倒排索引,对于某个单词来说,Map 阶段计算这个单词对其倒排索引文档集合中任意两个文档相似性的贡献,如果倒排索引中包含 m 个文档,则需要进行 $\frac{1}{2}m*(m-1)$ 次计算。本阶段输出的 Key 为文档,Value 为 $w_{t,i}*w_{t,j}$。Reduce 阶段则对相同文档对的所有 Value 值进行累加,即累加不同单词对两个文档相似性计算的贡献,这样即可得出两个文档的整体相似性。

通过以上方式即可对大规模文本计算集合计算其两两相似性。实验表明,文档大小和计算时间基本呈线性增长关系,所以这是一种可扩展的计算方法,但是这种计算方式也有问题,在计算文档相似性的第二阶段 MR 任务的过程中,Map 阶段的输出存在组合爆炸问题,会产生大量的中间结果文件,从而占用了大量的磁盘空间,同时降低了计算效率。为了减少中间文件数量,可以采用 DF-Cut 的优化方法,即把某些停用词或者常用词进行过滤,这样可以极大地减少中间结果文件的大小及 Reduce 阶段的计算量,同时因为 DF 值大的单词往往意味着较少的信息含量,即使将其过滤,对于相似性计算结果也不会有很大的影响。

13.2.4 阿里云 PAI 与 ET

阿里云 PAI(Platform of Artificial Intelligence)是阿里(即阿里巴巴集团)的智能平台,其目的是为了加速整个创新过程,提高工作效率。该平台是基于阿里云的云计算平台,具有处理超大规模数据的能力和分布式的存储能力,同时整个模型支持超大规模的建模以及 GPU 计算。此外,该平台还具有社区的特点:实验结果可共享,社区团队相互协作。该智能平台主要分为 3 层,第一层是 Web UI 界面,第二层是 IDST 算法层,最后一层是 MaxCompute 平台层。

PAI 的特点如图 13-1 所示,其主要特点有适合大数据分析、具有分布式计算的架构、集成大量高级算法、具备完整的数据挖掘功能和良好的图形界面。

该平台通过交互的界面降低了技术门槛,使用者可以轻松实现数据挖掘的工作,而无需太多经验;其次,其内嵌的算法都经过阿里内部多年的淬炼,在性能和准确率上都有较大的提升;最后是数据智能,该平台提供了从元数据到模型部署的整套流程,通过提供基本的组件,使用者可以搭建各个垂直场景下的解决方案。

其客户主要包括以下几类:一类是传统的大型企业和政府部门,如中石化、中石油、气象局等;另一类是中小企业,主要是公共云上的初创用户;以及一些个人用户,如数据科学家、研究人员等。

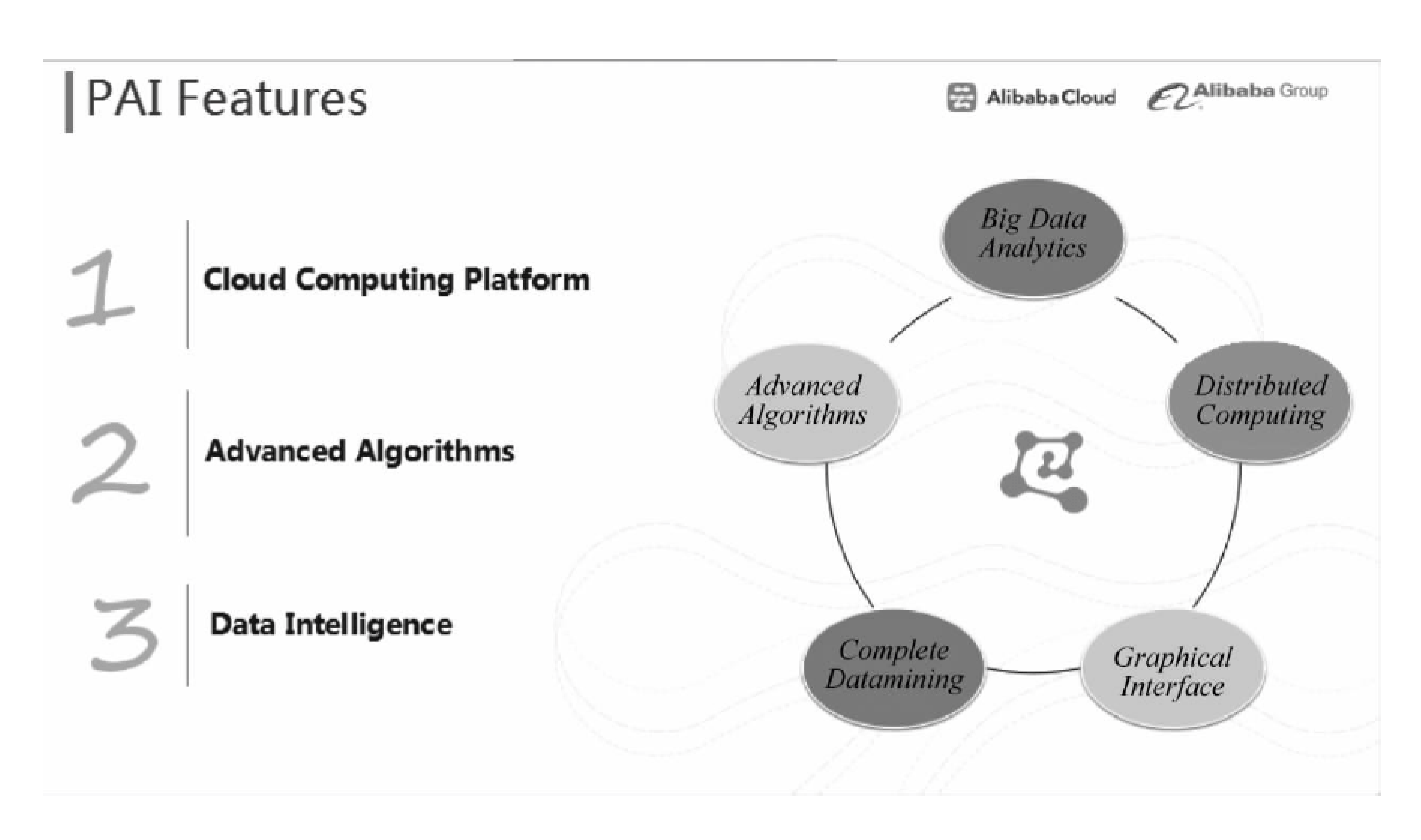


图 13-1 PAI 平台的特点

图 13-2 是 PAI 整体框架图，最底层是基础设施层，包括 CPU 和 GPU 集群；其上一层是阿里提供的计算框架，包括 MapReduce、SQL、MPI 等计算方式；中间一层是模型算法层，包含数据预处理、特征工程、机器学习算法等基本组件，帮助使用者完成简单的工作；平台化产品层主要是项目管理、算法模型分享，以及一些特定的需求；最上层是应用层，阿里内部的搜索、推荐、蚂蚁金服等项目在进行数据挖掘工作时都是依赖 PAI 平台产品。



图 13-2 PAI 整体框架图

阿里云 ET 则是基于阿里的人工智能平台研发的人工智能，其特色在于基于强大的云计算和大数据处理能力，目前 ET 具备语音识别、图像/视频识别、交通预测、情感分析等技能，并朝着大数据 AI 的方向发展。ET 的前身是阿里云小 AI，它曾在湖南卫视“我是歌手”节目中准确预测决赛歌王归属李玟，推动人工智能技术向理解人类情感迈出了重要一步。

13.3 深度学习

13.3.1 深度学习简介

深度学习(Deep Learning)是机器学习的分支，它试图使用包含复杂结构或由多重非线性变换构成的多个处理层对数据进行高层抽象的算法。深度学习是机器学习中表征学习的方法。观测值(如一幅图片)可以使用多种方式来表示，如每个像素强度值的向量，或者更抽象地表示成一系列边、特定形状的区域等。而使用某些特定的表示方法更容易从实例中学习任务。深度学习的好处是将用非监督式学习或半监督式的特征学习和分层特征提取的高效算法来替代手工获取特征。

深度学习框架，尤其是基于人工神经网络的框架可以追溯到 1980 年福岛邦彦提出的新认知机，而人工神经网络的历史更为久远。1989 年，燕乐存(Yann LeCun)等人开始将 1974 年提出的标准反向传播算法应用于深度神经网络，这一网络被用于手写邮政编码的识别。尽管该算法可以成功执行，但计算代价非常巨大，神经网络的训练时间达到了 3 天，因而无法投入实际使用。许多因素导致了这一缓慢的训练过程，其中一种是由于尔根・施密德胡伯(Jürgen Schmidhuber)的学生赛普・霍克赖特(Sepp Hochreiter)于 1991 年提出的梯度消失问题。与此同时，神经网络也受到了其他更加简单的模型的挑战，支持向量机等模型在 20 世纪 90 年代到 21 世纪初成为更加流行的机器学习算法。

“深度学习”这一概念从 2007 年前后开始受到关注，当时，杰弗里・辛顿(Geoffrey Hinton)和鲁斯兰・萨拉赫丁诺夫(Ruslan Salakhutdinov)提出了一种在前馈神经网络中进行有效训练的算法。这一算法将网络中的每一层视为无监督的受限玻尔兹曼机，再使用有监督的反向传播算法进行调优。在此之前的 1992 年，在更为普遍的情形下，施密德胡伯也曾在递归神经网络上提出一种类似的训练方法，并在实验中证明这一训练方法能够有效地提高有监督学习的执行速度。

自深度学习出现以来，它已成为很多领域(尤其是计算机视觉和语音识别)各种领先系统的一部分。在通用的用于检验的数据集(如语音识别中的 TIMIT 和图像识别中的

ImageNet、Cifar10)上的实验证明,深度学习能够提高识别的精度。

硬件的进步也是深度学习重新获得关注的重要因素。高性能图形处理器的出现极大地提高了数值和矩阵运算的速度,使得机器学习算法的运行时间得到了显著的缩短。

13.3.2 DistBelief

实验表明,当增加深度神经网络的规模(包括增加训练数据和模型参数规模)时,深度学习模型的分类精度会有大幅度提高。但是当模型的训练数据的规模达到千万量级、模型参数达到10亿量级时,GPU、MR等方式的并行计算架构都无法有效地解决此等规模的问题。为了能够解决如此规模的机器学习应用问题,Google开发了DistBelief分布式计算框架,在本质上,DistBelief是能够同时支持数据并行和模型并行的参数服务器架构。

SGD是深度学习模型训练过程最常用的参数优化方法,下面介绍DistBelief参数服务器框架下的两个并行训练方法,其中一个是在线学习(Online Learning)方法,被称为Downpour SGD,另一个是批学习(Batch Learning)版本,被称为Sandblaster L-BFGS。

SGD作为一个在线学习模型,本质上是一个串行算法,需要依次根据训练数据更新模型参数,而这对于大规模的训练数据在效率上是无法接受的。Downpour SGD可以提高训练效率,其本质是采用Mini-Batch方式更新参数的异步SGD模型。

首先将训练数据划分成若干子集合,每个子集合各自进行模型训练,子集合当前正在训练的模型可以成为模型副本。在训练过程中,副本模型通过和保存全局模型参数的参数服务器通信来获得全局参数或者通知其进行参数更新。参数服务器本身也是服务器集群,并对全局参数进行数据分片,每个数据分片负责一部分全局参数的存储与数值更新。之所以说Downpour SGD是异步的,其含义有两个方面:一方面是每个副本模型各自异步地执行;另一方面,参数服务器的数据分片之间也是相互独立,无须同步更新的。通过双方面的异步执行有效地加快了训练速度。

在每个副本模型利用训练子集合进行Mini-Batch更新参数之前,其首先从参数服务器获取当前的全局模型参数。因为DistBelief支持模型并行,即每个副本模型本身也是分布到多台机器上的,所以每台机器只需要和对应存储的与自身参数一致的参数服务器进行通信即可得到全局模型参数,这样可以有效地减少数据传输量。在接收到全局模型参数之后,副本模型运行一次Mini-Batch更新,计算局部梯度,然后将局部梯度传给对应的参数服务器,参数服务器根据局部梯度来更新全局模型参数。通过设定每隔n_{fetch}步来进行一次拉取全局模型参数操作以及设定每隔n_{push}步来进行一次局部梯度数据操作,可以有效地控制通信总量。与标准的同步SGD相比,异步SGD有更好的容错性。当某个机器发生故障时并不影响其他机器异步更新全局模型参数。尽管无法从理论上保证异

步 SGD 的正确性，但是实验效果表明，其计算精度可以达到与类似算法相近的效果，同时运行效率也获得了极大的提升。

13.3.3　TensorFlow

TensorFlow 是 Google 公司于 2015 年发布的机器学习平台，发布以后由于速度快、扩展性好，推广速度很快。

TensorFlow，从名字来看就是张量的流动。张量(tensor)即任意维度的数据，一维、二维、三维、四维等数据统称为张量。张量的流动则是指保持计算节点不变，让数据进行流动。这样的设计是针对连接式的机器学习算法，如逻辑回归、神经网络等。连接式的机器学习算法可以把算法表达成一张图，张量从图中从前到后走一遍就完成了前向运算；而残差从后往前走一遍就完成了后向传播。

在 TF 的实现中，机器学习算法被表达成图，图中的节点是算子(operation)，节点会有 0 到多个输出，表 13-1 是 TF 实现的一些算子。

表 13-1　TF 实现的算子

类　　别	例　　子
Element-wise mathematical operations	Add、Sub、Mul、Div、Exp、Log、Greater、Less、Equal 等
Array operations	Concat、Slice、Split、Constat、Rank、Shape、Shuffle 等
Matrix operations	MatMul、MatrixInverse、MatrixDeterminant 等
Stateful operations	Variable、Assign、AssignAdd 等
Neural-net building blocks	SoftMax、Sigmoid、ReLU、Convolution2D、MaxPool 等
Checkpointing operations	Save、Restore
Queue and synchronization operations	Enqueue、Dequeue、MutexAcquire、MutexRelease 等
Control flow operations	Merge、Switch、Enter、Leave、NextIteration

每个算子都会有属性，所有的属性都在建图的时候被确定下来，最常用的属性是为了支持多态，比如加法算子既能支持 float32 计算，又能支持 int32 计算。

TF 中还有一个概念是 kernel，kernel 是 operation 在某种设备上的具体实现。TF 的库通过注册机制来定义 op 和 kernel，所以可以通过链接一个其他库进行 kernel 和 op 的扩展。

TF 的图中的边分为下面两种。

(1) 正常边。正常边上可以流动数据，即正常边就是 tensor。

(2) 特殊边。特殊边又称控制依赖(control dependencies)。

- 没有数据从特殊边上流动，但是特殊边却可以控制节点之间的依赖关系，在特殊边的起始节点完成运算之前，特殊边的结束节点不会被执行。
- 不仅仅非得有依赖关系才可以用特殊边，还可以有其他用法，例如控制内存的时候可以让两个实际并没有前后依赖关系的运算分开执行。
- 特殊边可以在 client 端被直接使用。

客户端使用会话和 TF 系统交互。一般的模式是：建立会话，此时会生成一张空图；在会话中添加节点和边，形成一张图，然后执行。

下面是一个 TF 的会话示例，图 13-3 是对应的图示。

```
import tensorflow as tf
b = tf.Variable(tf.zeros([100]))                       # 100 - d vector, init to zeroes
W = tf.Variable(tf.random_uniform([784,100], - 1,1))   # 784x100 matrix w/rnd vals
x = tf.placeholder(name = "x")                         # Placeholder for input
relu = tf.nn.relu(tf.matmul(W,x) + b)                  # Relu(Wx + b)
C = [...]                                              # Cost computed as a function
                                                       # of Relu

s = tf.Session()
for step in xrange(0,10):
  input = ...construct 100 - D input array...          # Create 100 - d vector for input
  result = s.run(C, feed_dict = {x:input})             # Fetch cost, feeding x = input
  print step, result
```

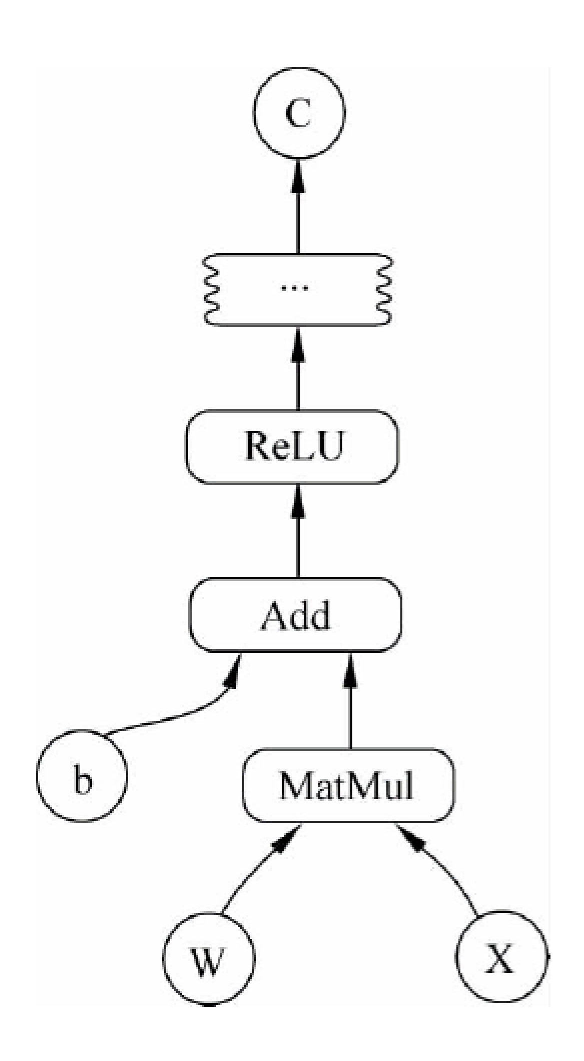


图 13-3　TF 会话图示

机器学习算法都会有参数，而参数的状态是需要保存的。参数在图中有其固定的位置，不能像普通数据那样正常流动。因而，在 TF 中将 Variables 实现为一个特殊的算子，

该算子会返回它所保存的可变 tensor 的句柄。

TF 的实现分为以下几个部分：

(1) TF 中对最重要的 tensor 支持得非常全面，从 8bit 到 64bit，signed 和 unsigned，IEEE float/double，complex number 等。使用引用计数来保存 tensor，当计数到 0 时 tensor 被回收。

(2) 客户端。用户会使用，与 master 和一些 worker process 交流。

(3) master。其用来与客户端交互，同时调度任务。

(4) worker process。即工作节点，每个 worker process 可以访问一到多个 device。

(5) device。即 TF 的计算核心，通过 device 的类型、job 名称、在 worker process 中的索引对 device 命名，可以通过注册机制来添加新的 device 实现，每个 device 实现需要负责内存分配和管理调度 TF 系统所下达的核运算需求。

TF 的实现分为单机实现和分布式实现。在分布式实现中，需要实现的是对 client、master、worker process 不在同一台机器上时的支持。分布式和单机的不同架构如图 13-4 所示。

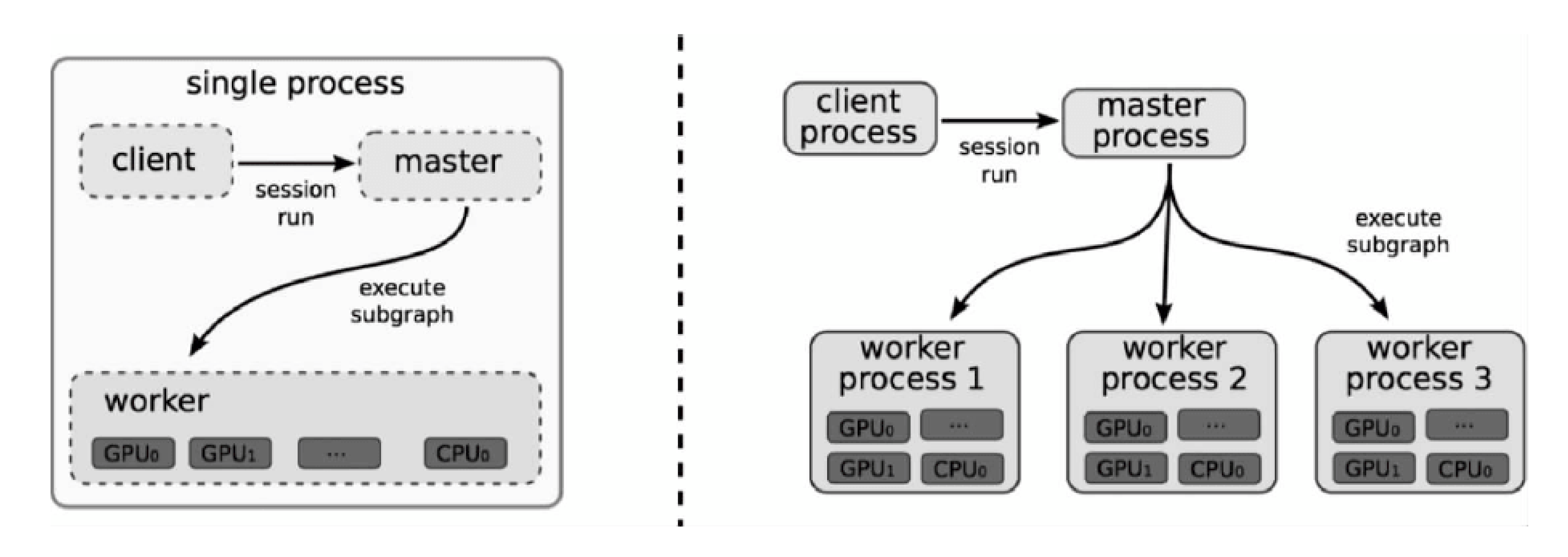


图 13-4 TF 的单机与分布式架构

在 TF 出现之前已经有很多类似的平台，例如 Theano、Torch、Caffe、Chainer、Computational Network。

可以说，TF 从它每一个平台中都吸取了一些特性和设计思想。TF 与其他平台的区别和联系如下。

(1) 支持符号推导，如 Theano 和 Chainer。

(2) 使用 C++ 写内核，从而方便跨平台部署，如 Caffe。

(3) 支持跨设备计算，让用户高层表达模型，如 Adam 和 DistBelief，但比 Adam 和 DistBelief 更优越的是更有弹性，支持更多的模型。

(4) 相对于 Adam、DistBelief 和 Parameter Server，TF 把参数节点化，更新操作也是图中的一个节点，而 Adam 等 3 个会有一个独立的 Parameter Server。

(5) Halide 拥有和 TF 相似的中间表达，却有更高级的语义表示，在并行方面的优化

更多，但却是单机的，TF 有意向此方向发展，将 Halide 的特性添加到 TF 中。

13.4 数据挖掘与机器学习的发展趋势

随着数据挖掘与机器学习领域的研究不断深入和广泛，人们关注的焦点也有了新的变化，总的趋势是数据挖掘与机器学习的研究和应用更加“社会化”和“大数据化”。数据挖掘与机器学习的理论和应用在相当一段时间继续保持稳定发展，但有着朝向大规模的社交数据分析和时间序列数据分析方向发展的趋势。在用户层面，移动计算设备的普及与大数据革命带来的机遇使得搜索引擎对用户所处的上下文环境具有了前所未有的深刻认识，但对于如何将认识上的深入转化为用户信息获取过程的便利仍然缺乏成功经验。除此之外，社交网络服务的兴起对互联网数据环境和用户群体均将形成关键性的影响，如何更好地面对相对封闭的社交网络数据环境和被社交关系组织起来的用户群体也是数据挖掘面临的机遇和挑战。可以预见会有更多的研究深入探讨数据挖掘的本质问题，不断完善其理论基础。同时，数据挖掘与机器学习的应用关注“大数据”的趋势越来越明显。在大规模数据下如何保证现有数据挖掘算法的时间和空间复杂度的应用成为研究热点。对于数据挖掘与机器学习技术，除了要求有好的效果之外，往往还要求其能够被高效地实现，以适用于更大规模、更高速演化的网络数据。此外，另一个研究热点将是网络安全、隐私以及软件可靠性和可扩展性。

13.5 习题

1. 简述批量计算与流式计算的区别。
2. 简述交互式数据处理的应用场景。
3. 简要介绍机器学习的分类。
4. 简要介绍监督学习中用于度量预测的错误程度的函数。
5. 对于广告点击率问题，使用逻辑回归算法来确定广告展出与否以及广告展出的顺序。
6. 简述如何对大规模文档进行相似性计算。
7. 简述并行随机梯度下降算法。

第 14 章

大数据实践：基于数加平台的推荐系统

14.1 数据集简介

阿里移动推荐数据竞赛是以阿里巴巴移动电商平台的真实用户-商品行为数据为基础，同时提供了移动时代特有的位置信息。目前可以在 tianchi.shuju.aliyun.com 的天池新人实战赛之“平台赛”找到该数据集，该数据集提供了 100 万用户的完整行为数据以及百万级的商品信息。其中训练数据包含了抽样出来的一定量用户在一个月时间(11.18～12.18)之内的移动端行为数据(D)，评分数据是这些用户在这一个月之后的一天(12.19)对商品子集(P)的购买数据。参赛者要使用训练数据建立推荐模型，并输出用户在接下来的一天对商品子集购买行为的预测结果。

在数加平台上，研究者们可以基于此数据集通过大数据分析工具和算法平台构建面向移动电子商务的商品推荐模型。

当进入自己所在的项目后，根据赛题数据的描述可以读取或者复制对应的表到自己的项目空间。为了方便操作，在自己的项目空间执行以下 SQL 语句，可以复制相关表到自己的项目空间。

```
CREATE TABLE IF NOT EXISTS tianchi_fresh_comp_train_user_online AS SELECT * FROM odps_tc_
257100_f673506e024.tianchi_fresh_comp_train_user_online;
```

```
CREATE TABLE IF NOT EXISTS tianchi_fresh_comp_train_item_online AS SELECT * FROM odps_tc_
257100_f673506e024.tianchi_fresh_comp_train_item_online;
```

14.2 数据探索

使用简单的 SQL 语句即可对数据进行查看，如图 14-1 所示。

更多功能

test

```
select * from tianchi_fresh_comp_train_user_online;
```

日志 结果[1] 结果[2]

序号	user_id	item_id	behavior_type	user_geohash	item_category	time
1	85764880	151364310	1	9q1wbdg	10392	2014-12-06 23
2	85764880	383479051	1	9q1wbgp	5894	2014-12-07 22
3	85764880	247955935	1	\N	13729	2014-12-01 19
4	85764880	208173708	1	9q1iit5	675	2014-12-16 23
5	85764880	123111134	1	9q1wbf3	8077	2014-12-14 23
6	85764880	374666510	1	9q1wbf7	675	2014-12-16 23
7	85764880	211023354	1	9q1r3on	9516	2014-11-18 21
8	85764880	322975728	1	9q1wbf6	8077	2014-12-14 23
9	85764880	76117661	1	9q1wbfa	5232	2014-12-06 23
10	85764880	392685393	1	9q1iitr	4370	2014-12-06 23
11	85764880	76117661	1	9q1wbfw	5232	2014-12-06 23
12	85764880	158327788	1	\N	5271	2014-11-18 23
13	85764880	21899384	1	\N	484	2014-12-10 22

图 14-1　用 SQL 语句查看数据

图 14-2 展示了一个统计用户数量、商品数量以及有过交互的用户-商品对数量的例子，可以看到在用户行为数据表中用户数量为 100 万个，商品数量为 66 997 095 个，有过交互的用户-商品对数量为 444 526 713 个。这样总的用户-商品对数量为 $6.699\,709\,5\times 10^{13}$ 个。



图 14-2　所有用户-商品对结果

图 14-3 则只统计有过购买行为的用户数量、商品数量以及用户-商品对数量的例子，可以看到有过购买行为的用户数量为 879 217 万个，商品数量为 4 237 620 个，用户-商品对数量为 9 852 779 个。可以看到，有过购买行为的用户-商品对数量与总的用户-商品对数量相比，数量十分悬殊，即使是和有过交互行为的用户-商品对相比，其比例也仅占 2%，即购买行为十分稀疏。



图 14-3　有过购买行为的商品-用户对结果

使用算法平台的统计分析控件可以对数据进行简单的可视化处理。例如图 14-4 是使用直方图控件观察不同类型的行为数量分布的例子，4 种类型的行为分别对应浏览、收藏、加购物车和购买。

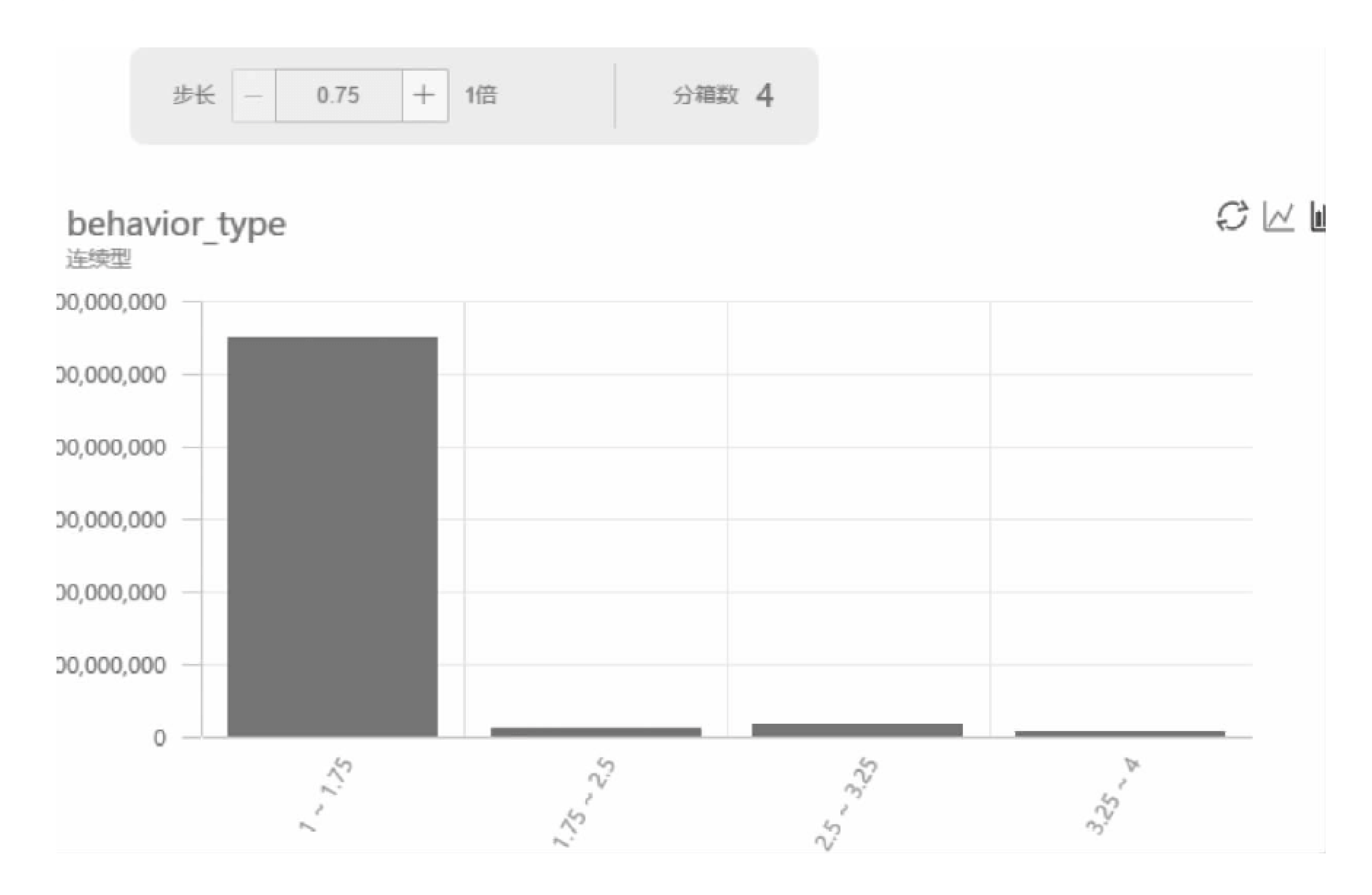


图 14-4　不同类型的行为数量分布

图 14-5 是使用直方图控件观察每天行为数量分布的例子，可以看到在双 12 当天用户行为数量出现了一个高峰。

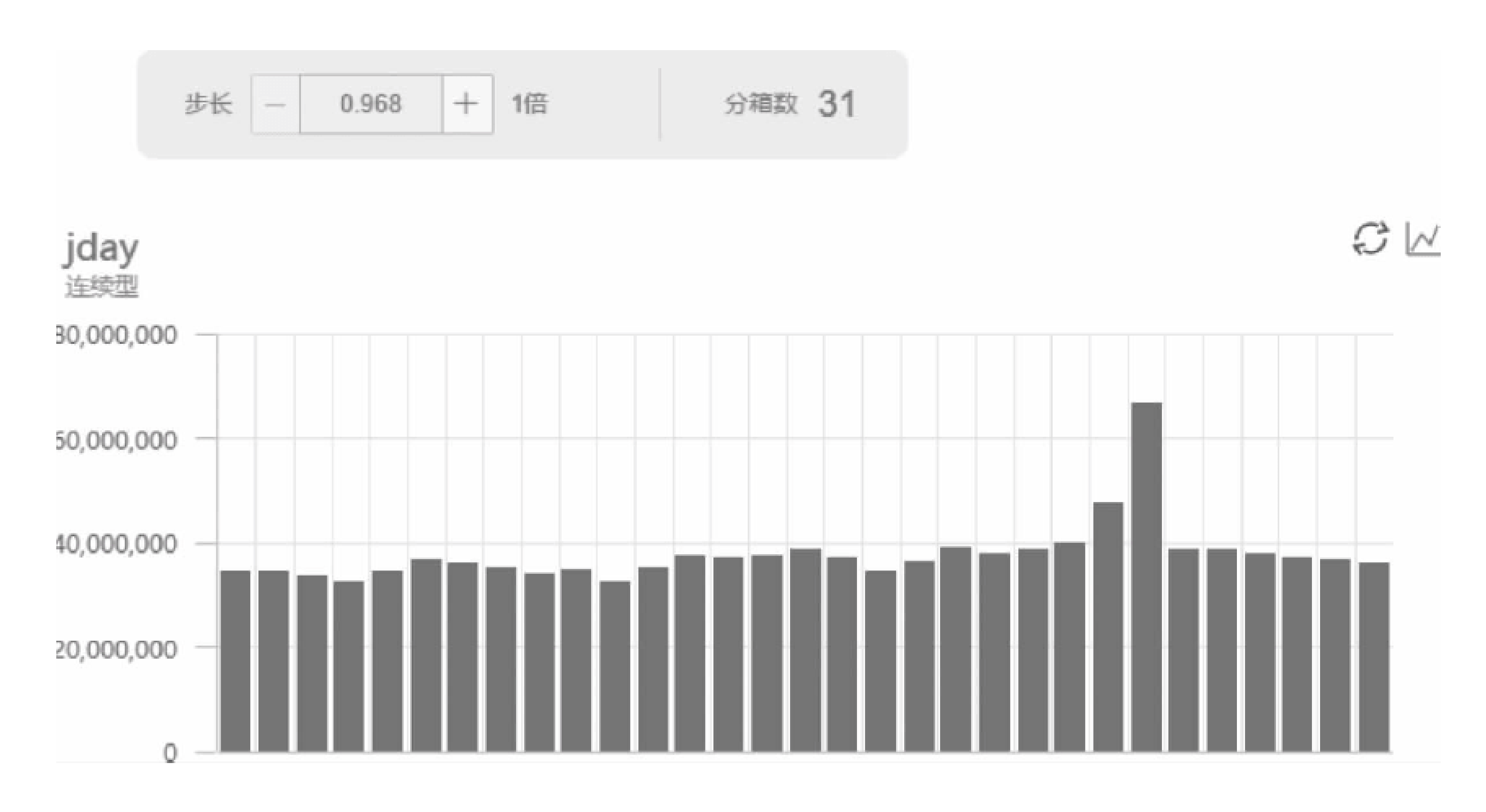


图 14-5　每天行为数量分布

14.3　方案设计

预测用户未来的购买行为是推荐实施过程中的关键步骤，预测的准确率将直接影响到推荐的效果。如何提高对购买行为预测的准确率一直是学术界和工业界关注的话题。对于每一个用户-商品对来说，预测其未来是否会发生购买行为正好可以对应到机器学习领域的二分类问题，从而可以采用机器学习中的二分类模型，根据用户的历史行为数据对用户和商品进行建模，对用户未来的购买行为进行预测。

14.4　训练集构造

14.4.1　MapReduce 环境配置

要训练一个二分类模型首先需要得到训练集，而产生训练集需要提取特征和标签。在数加平台上进行特征提取可以使用 MaxCompute SQL、MapReduce 以及 UDF 函数，其中编写 MapReduce 程序提取特征相对于 SQL 更加灵活，下面简要介绍 MapReduce 的环境配置。

配置 MapReduce 的环境通常需要 Maven 的支持，安装 Maven 首先需要下载并解压 apache-maven-3.3.3-bin.zip，然后新建环境变量 M2_HOME 到解压目录下的 bin 目录，设置完成后在 CMD 运行 mvn -v 可以观察配置是否成功，安装成功的信息如图 14-6 所示。

```
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

C:\Users\lenovo>mvn -v
Apache Maven 3.3.3 (7994120775791599e205a5524ec3e0dfe41d4a06; 2015-04-22T19:5
7+08:00)
Maven home: D:\eclipseHadoop1x\apache-maven-3.3.3
Java version: 1.7.0_51, vendor: Oracle Corporation
Java home: C:\Program Files\Java\jdk1.7.0_51\jre
Default locale: zh_CN, platform encoding: GBK
OS name: "windows 7", version: "6.1", arch: "amd64", family: "windows"
```

图 14-6 mvn 安装成功示意图

最后安装 Eclipse 的 Maven 插件即可开始创建项目，对于详细的配置过程和代码编写可以参考《天池用户手册_数加平台》(在阿里云论坛搜索"数加平台指南"即可找到该手册)。

14.4.2 MapReduce 代码编写

下面以统计每一个用户-商品对的 4 种行为的数量为例来说明如何编写一个 MapReduce 程序。在本例中，用户程序将用户行为表作为输入表，并产生一个结果表保存每一个用户-商品对的 4 种行为的数量。输入表用到的列名有 user_id、item_id、behavior_type，结果表的列名有 user_id、item_id、ui_bro_cnt、ui_fav_cnt、ui_cart_cnt、ui_buy_cnt。

在 MyMapper 类中，用户代码从输入表中读取用户 ID、商品 ID、操作类型信息，并将用户 ID 和商品 ID 作为 OutputKey，将 4 种行为的统计数量分别命名为 ui_bro、ui_fav、ui_cart、ui_buy，并作为 OutputValue。OutputKey 和 OutputValue 均传给 Reduce Worker 做进一步处理，具有相同 OutputKey 的一条记录会分配给同一个 Reduce Worker。

MyReducer 在对某个键值(OutputKey)的记录进行统计之后将键值对应的用户 ID、商品 ID 以及统计结果输出到结果表。所有的实现代码如下：

```
package Feature.Count;



import java.io.IOException;
import java.util.Iterator;

import com.aliyun.odps.data.Record;
import com.aliyun.odps.data.TableInfo;
import com.aliyun.odps.mapred.JobClient;
```

```
import com.aliyun.odps.mapred.MapperBase;
import com.aliyun.odps.mapred.ReducerBase;
import com.aliyun.odps.mapred.conf.JobConf;
import com.aliyun.odps.mapred.utils.InputUtils;
import com.aliyun.odps.mapred.utils.OutputUtils;
import com.aliyun.odps.mapred.utils.SchemaUtils;


public class Count {

  public static class MyMapper extends MapperBase {
    private Record key;
    private Record value;

    @Override
    public void setup(TaskContext context) throws IOException {
        key = context.createMapOutputKeyRecord();
        value = context.createMapOutputValueRecord();
    }

    private void initRecord(Record outputValue)
    {
        outputValue.setBigint("ui_bro", 0L);
        outputValue.setBigint("ui_fav", 0L);
        outputValue.setBigint("ui_cart", 0L);
        outputValue.setBigint("ui_buy", 0L);
    }

    @Override
    public void map(long recordNum, Record record, TaskContext context)
        throws IOException {
        //初始化值
        initRecord(value);

        //获取信息
        String user_id = record.getString("user_id");
        String item_id = record.getString("item_id");

        long act_type = record.getBigint("behavior_type");
```

```
            int iact = (int)act_type;

            //设置键记录
            key.setString("user_id", user_id);
            key.setString("item_id", item_id);

            switch(iact)
            {
            //在这里添加浏览操作功能
            case 1:
                value.setBigint("ui_bro", 1L);
                break;
            //在这里添加喜欢的动作特性
            case 2:
                value.setBigint("ui_fav", 1L);
                break;
            //在这里添加购物车操作功能
            case 3:
                value.setBigint("ui_cart", 1L);
                break;
            //添加购买行为特性
            case 4:
                value.setBigint("ui_buy", 1L);
                break;
            default:
                break;
            }

            context.write(key, value);
            context.progress();
        }
    }

    /**
     * A reducer class that just emits the sum of the input values.
     **/
    public static class MyReducer extends ReducerBase {
      private Record result;
```

```
        @Override
        public void setup(TaskContext context) throws IOException {
            result = context.createOutputRecord();
        }

        @Override
        public void reduce(Record key, Iterator<Record> values, TaskContext context)
            throws IOException {
            //输出的关键
            result.setString("user_id", key.getString("user_id"));
            result.setString("item_id", key.getString("item_id"));
            //action num counting variable
            long bro_cnt = 0L, fav_cnt = 0L, cart_cnt = 0L, buy_cnt = 0L;
            while(values.hasNext())
            {
                Record val = values.next();
                //计数操作数
                bro_cnt += val.getBigint("ui_bro");
                fav_cnt += val.getBigint("ui_fav");
                cart_cnt += val.getBigint("ui_cart");
                buy_cnt += val.getBigint("ui_buy");
            }
            //过滤非交互式
            if(bro_cnt + fav_cnt + cart_cnt + buy_cnt > 0)
            {
                result.setBigint("ui_bro_cnt", bro_cnt);
                result.setBigint("ui_fav_cnt", fav_cnt);
                result.setBigint("ui_cart_cnt", cart_cnt);
                result.setBigint("ui_buy_cnt", buy_cnt);

                context.write(result);
            }
            context.progress();
        }
    }

    public static void main(String[] args) throws Exception {
        if (args.length != 2) {
          System.err.println("Usage: WordCount <in_table><out_table>");
```

```
        System.exit(2);
      }

      JobConf job = new JobConf();

      job.setMapperClass(MyMapper.class);
      job.setReducerClass(MyReducer.class);

job.setMapOutputKeySchema(SchemaUtils.fromString("user_id:string,item_id:string"));
job.setMapOutputValueSchema(SchemaUtils.fromString("ui_bro:bigint,ui_fav:bigint,ui_
cart:bigint,ui_buy:bigint"));

      InputUtils.addTable(TableInfo.builder().tableName(args[0]).build(), job);
      OutputUtils.addTable(TableInfo.builder().tableName(args[1]).build(), job);

      JobClient.runJob(job);
    }

}
```

编译完成后，将其导出到JAR文件，并上传至数加平台的资源管理中，然后在任务开发中新建一个OPEN_MR的节点任务，对其做如下配置，如图14-7所示。



图14-7　OPEN_MR节点任务配置

最后运行该节点，得到各类行为的统计结果，如图 14-8 所示。

```
SELECT * FROM count_result ;
```

日志　结果[1]

序号	user_id	item_id	ui_bro_cnt	ui_fav_cnt	ui_cart_cnt	ui_buy_cnt
1	100000027	1732041	2	0	0	0
2	100000027	295100966	1	0	0	0
3	100000027	41418089	1	0	0	0
4	100000027	93973493	1	0	0	0
5	100000309	103820166	3	0	0	0
6	100000309	147714136	2	0	0	0
7	100000309	151039667	1	0	0	0
8	100000309	151479495	4	0	0	0
9	100000309	15175559	2	0	0	0
10	100000309	151841770	3	0	0	0

图 14-8　MR 程序的运行结果

14.4.3　特征提取与标签提取

如果要对每一个用户-商品对做二分类预测，首先需要考虑用户特征和商品特征，用户特征主要用来反映用户的活跃程度以及是否有在未来一天购物的意愿，可用的特征有最近的浏览行为次数、最近的加购物车行为次数、最近一次行为时间等；商品特征主要用来反映商品的热度，可用的特征有最近的被购买次数、被购买频率等；用户对该商品的行为同样值得关注，如最近是否浏览该商品、最近是否将该商品加入购物车、最近是否购买该商品等。

由于数据集中提供了商品类别信息，因此类别特征也可以反映商品的热度，如同类商品的被购买次数、同类商品的被购买频率，将商品特征与类别特征进行组合则可以得到商品在其同类别商品中的相对热度，如该商品在同类商品中的被购买比例、该商品在同类商品中的被浏览比例、该商品在同类商品中的被购买次数排名等。同时，用户对该类别商品的行为也间接地反映了用户是否还有意愿购买该商品，例如用户在购买了同类商品之后可能不再有购买该商品的意愿。

因此本文所采用的特征类型包括用户特征、商品特征、商品类别特征、用户对商品类别的行为特征、用户对商品的行为特征以及组合特征。

为了体现时间的影响，可以将特征分成不同的时间粒度进行统计，例如按小时划分为最近 1 小时、最近两小时、最近 3 小时、最近 6 小时等，按天划分为最近 1 天、最近 3 天、最近 7 天等。即按照所指定的不同大小的时间窗口得到多组统计量，再将其进行组合，

从而得到更加多样化的特征。

由于数据集中给定的用户行为数据的日期是从 2014 年 11 月 18 日到 2014 年 12 月 18 日，目标是预测 2014 年 12 月 19 日的用户购买行为。那么一个很自然的构造训练集的思路就是以 12 月 18 日 0 点作为分界面，12 月 18 日之前的 28 天用作特征提取，12 月 18 日当天的用户购买行为用作标签提取，从而得到线上训练集。然后将分界面向后移动一天，即采用 12 月 19 日之前的 28 天用作特征提取，构造出线上预测集。通过训练集训练出分类模型，再用模型对预测集中的每个样本进行预测得到最终的预测结果。

为了方便在线下进行参数调优和模型验证，构建验证集进行线下验证也是十分必要的。为此，本文构建了线下训练集和线下测试集来进行线下验证，它们分别使用 12 月 17 日 0 点和 12 月 18 日 0 点作为分界面，前 28 天用作特征提取，后 1 天用作标签提取。训练集和测试集的构造方式如下，相应图示如图 14-9 所示。

```
DROP TABLE IF EXISTS feats_train_off;
CREATE TABLE feats_train_off AS SELECT feats_v6_o2o. * from feats_v6_o2o where labelday =
29;          /* 线下训练集 */
DROP TABLE IF EXISTS feats_test_off;
CREATE TABLE feats_test_off AS SELECT feats_v6_o2o. * from feats_v6_o2o where labelday =
30;          /* 线下测试集 */
DROP TABLE IF EXISTS feats_train_on;
CREATE TABLE feats_train_on AS SELECT feats_v6_o2o. * from feats_v6_o2o where labelday =
30;          /* 线上训练集 */
DROP TABLE IF EXISTS feats_test_on;
CREATE TABLE feats_test_on AS SELECT feats_v6_o2o. * from feats_v6_o2o where labelday =
31;          /* 线上测试集 */
```

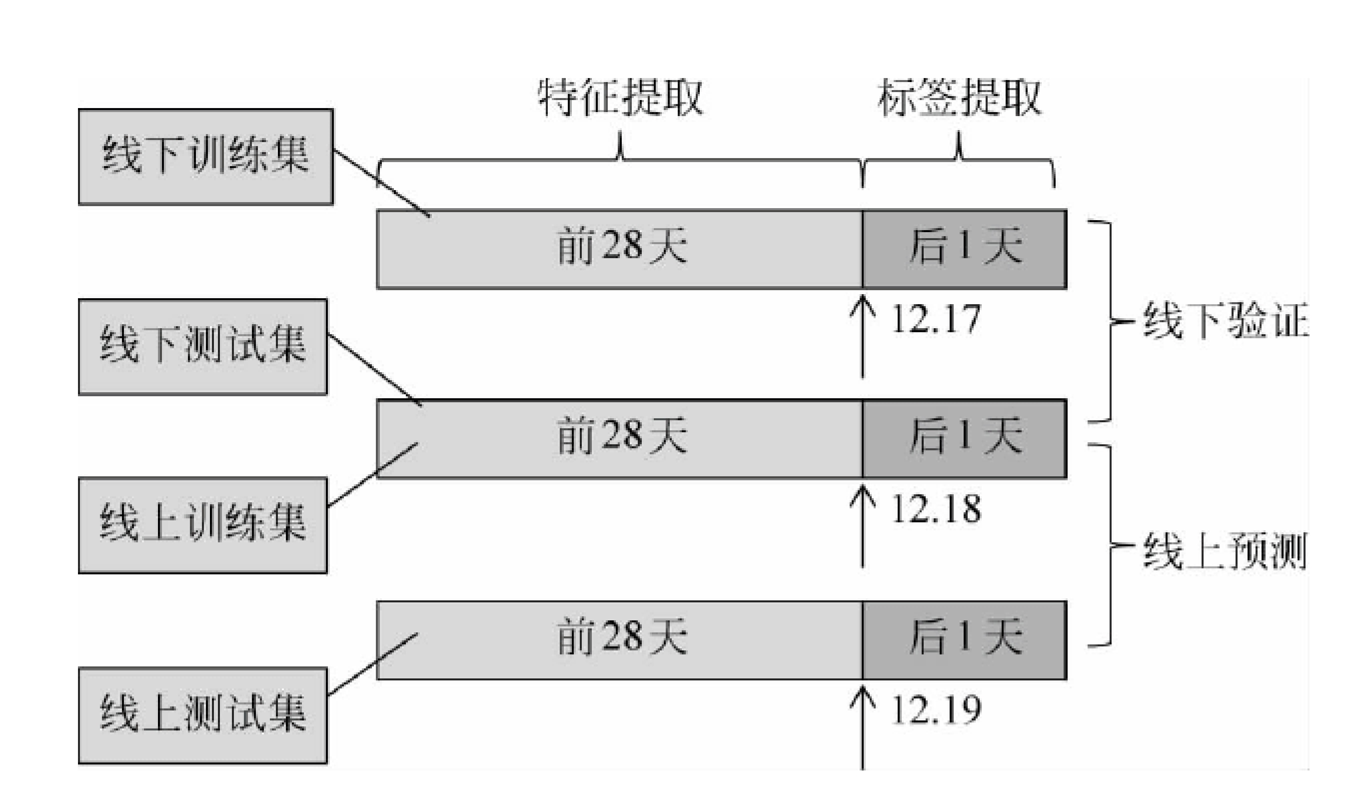


图 14-9　训练集和测试集构造图示

通过编写 MapReduce 程序可以提取特征和标签，并利用 Join 语句根据 user_id、item_id、item_category 对不同类型的特征进行连接，即可构造出训练集。构建训练集的整体流程如图 14-10 所示。

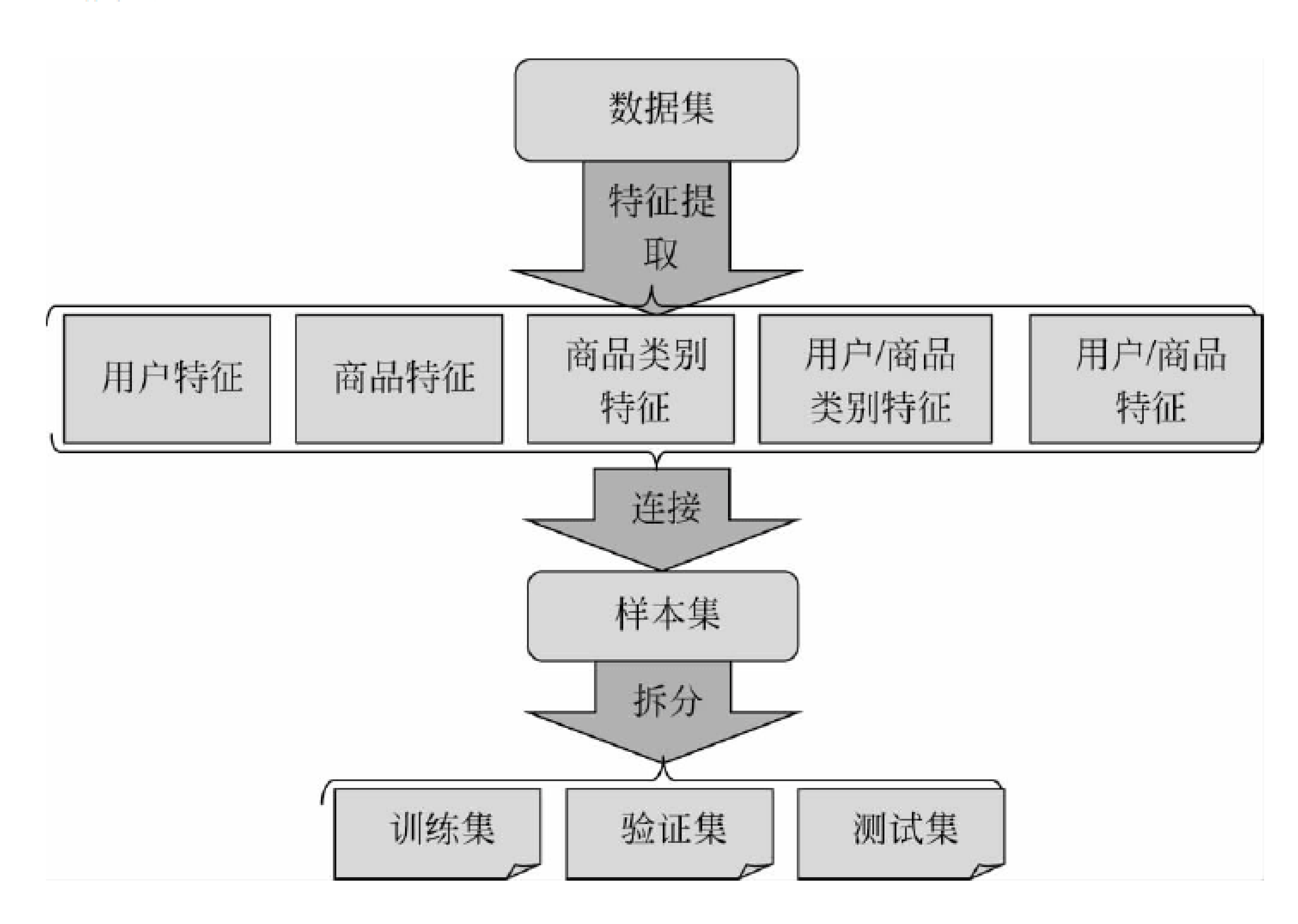


图 14-10　构建训练集的流程

14.4.4　训练集采样

在数据探索阶段可以发现购买行为十分稀疏，导致训练集存在正负样例不平衡的问题，过多的样本还会导致训练的效率十分低下。对此，一种简单的解决方式就是对负样本（未来没有购买行为的样本）使用随机下采样，从而调整类别不平衡度，并提高训练效率。

利用算法平台中的随机采样组件可以方便地对训练集进行随机采样，如图 14-11 所示。

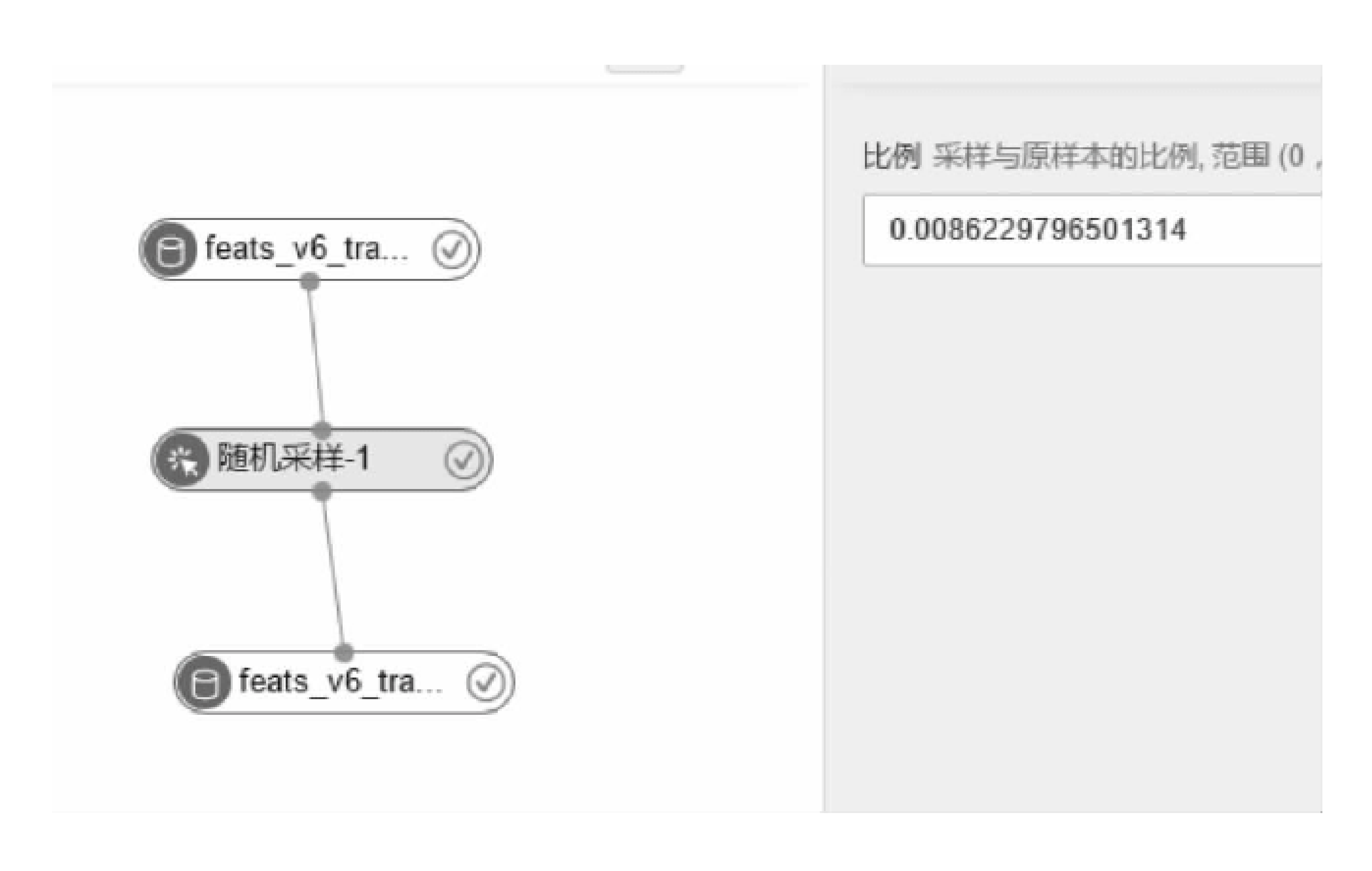


图 14-11　训练集随机采样

14.4.5　缺失值填充

在构造训练集的过程中，可能会遇到特征值为空或者 NAN 的情况，从而影响模型训练，例如对于算法平台的 Xgboost 模型，若特征值为 NAN 则在训练过程中会出现错误。使用算法平台的缺失值填充组件可以对缺失的特征值进行填充，如图 14-12 所示，将缺失值填充为 0，然后就可以正常进行模型训练。

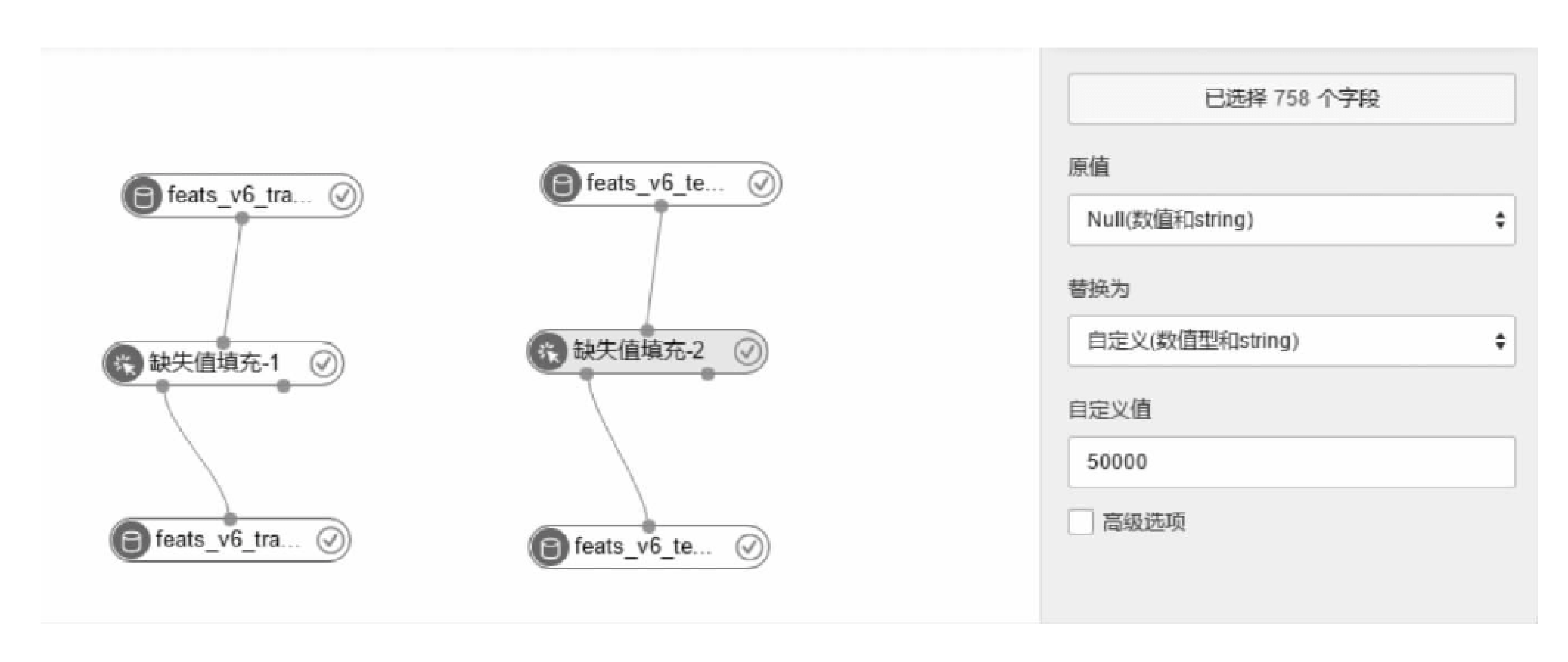


图 14-12　缺失值填充

14.5　模型训练与预测

在训练集构造完成之后就可以使用算法平台进行模型训练并产生预测结果。下面是一个训练 GBDT 模型并产生预测结果的例子。

首先从源/目标控件列表中拖出两个“读 ODPS 表”控件和一个“写 ODPS 表”控件，分别用于读取训练集、测试集以及存放预测结果，控件位置如图 14-13 所示。

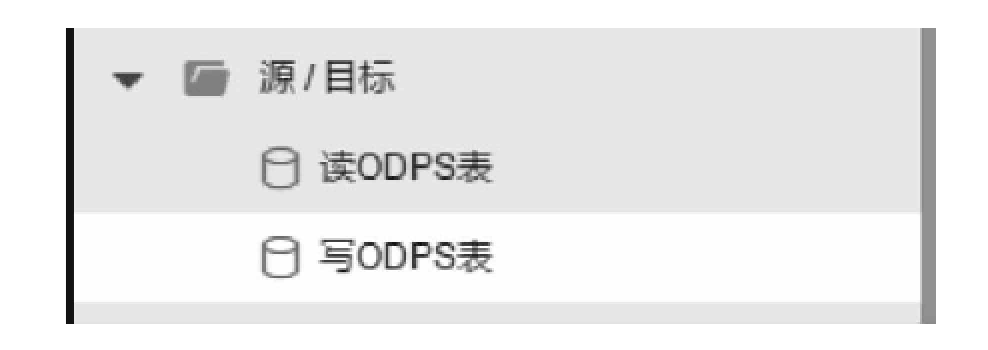


图 14-13　读/写 ODPS 控件

然后从机器学习下二分类的列表中找到“GBDT 二分类”控件，如图 14-14 所示。

再从机器学习的列表中找到“预测”控件，如图 14-15 所示。



图 14-14　GBDT 控件



图 14-15　预测控件

将控件排好位置，并进行连线，如图 14-16 所示。

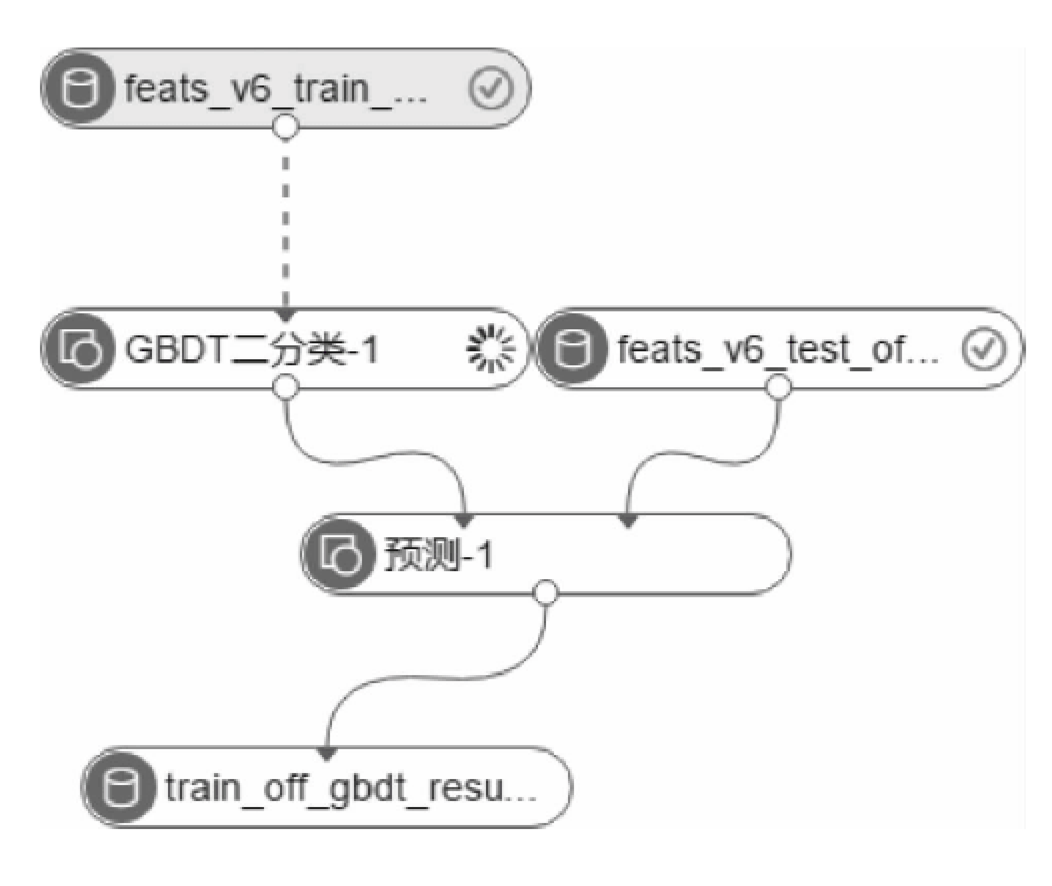


图 14-16　控件连线

左边输入的 ODPS 表是训练集，将其与 GBDT 控件相连，在 GBDT 控件中配置好相应的特征列、标签列以及相应的参数，如图 14-17 和图 14-18 所示，单击“运行”按钮即可开始进行模型训练。

右边输入的 ODPS 表是测试集，将 GBDT 控件和测试集一起与预测控件相连，在模型训练完成之后即可开始进行模型预测，并将预测结果写入最终的输出表。

在这里，预测控件生成的预测结果表为 train_off_gbdt_result，通过它可以产生推荐列表，例如通过以下 SQL 语句可以得到预测结果为 1 并且预测分数最高的前 1000 个推荐结果，生成的列表如图 14-19 所示。

```
SELECTuser_id, item_id, prediction_score FROM train_off_gbdt_result
WHERE prediction_result = 1 ORDER BY prediction_score DESC LIMIT 1000;
```



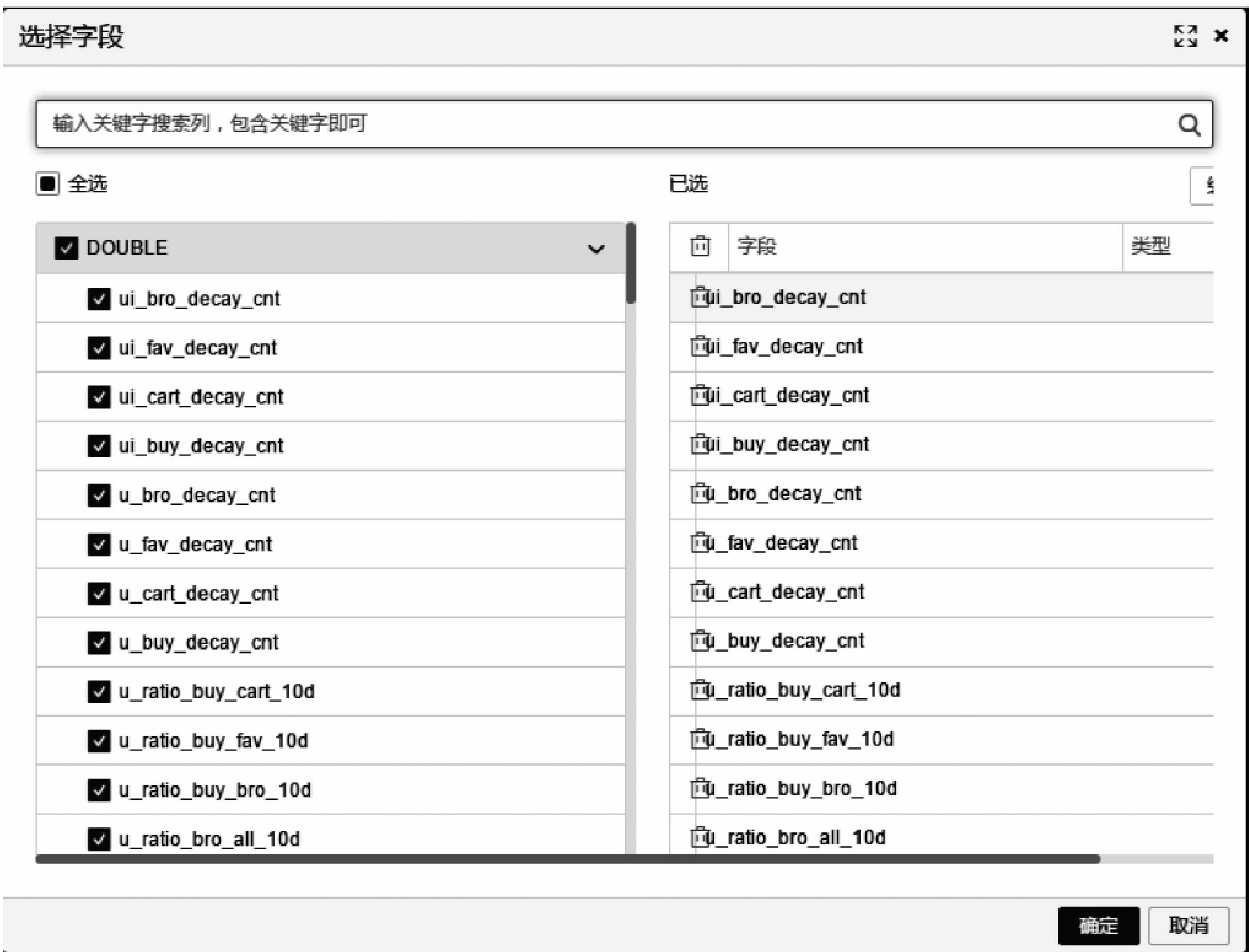


图 14-17　选择特征和标签

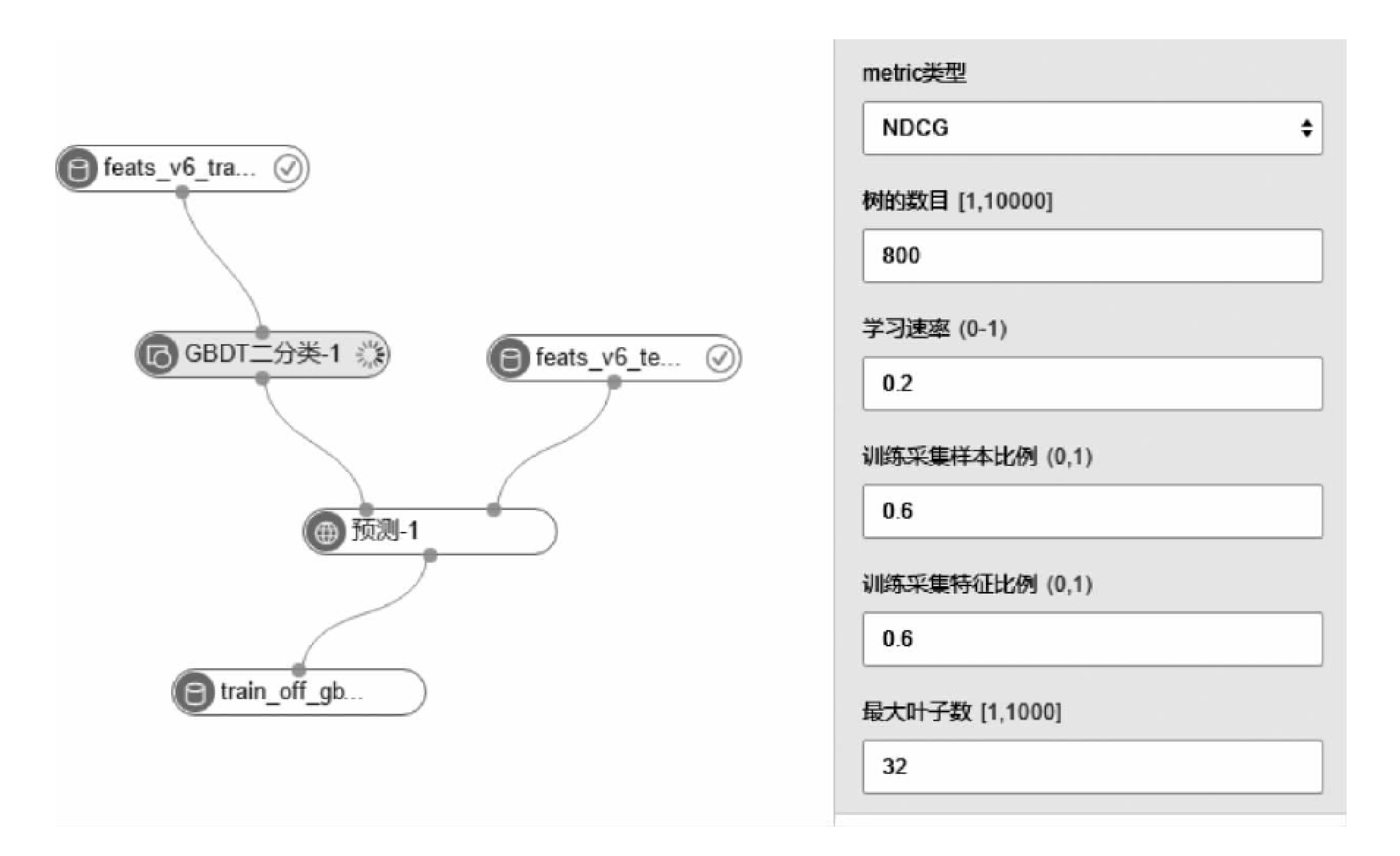
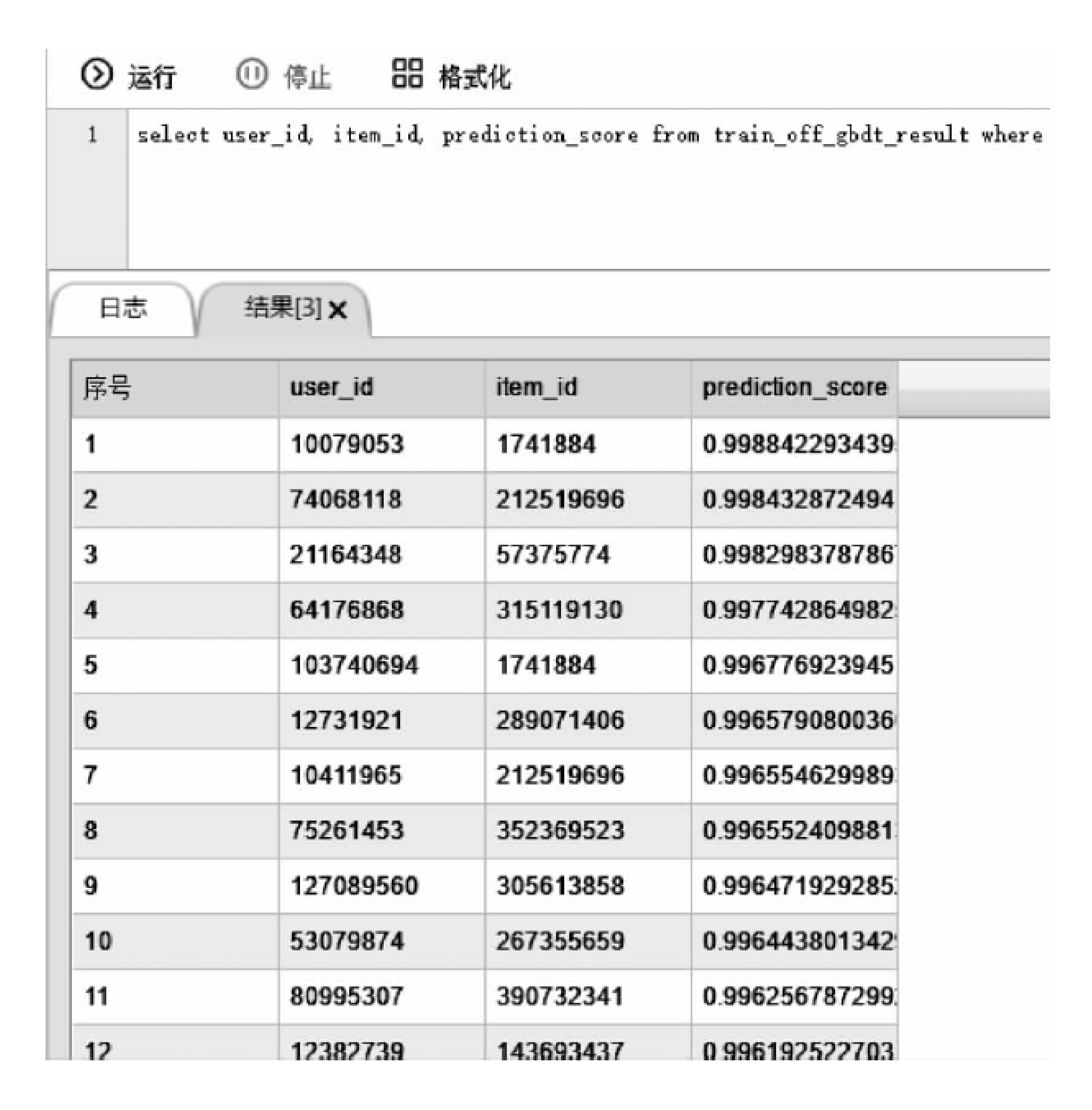


图 14-18 参数配置



序号	user_id	item_id	prediction_score
1	10079053	1741884	0.998842293439
2	74068118	212519696	0.998432872494
3	21164348	57375774	0.998298378786
4	64176868	315119130	0.997742864982
5	103740694	1741884	0.996776923945
6	12731921	289071406	0.996579080036
7	10411965	212519696	0.996554629989
8	75261453	352369523	0.996552409881
9	127089560	305613858	0.996471929285
10	53079874	267355659	0.996443801342
11	80995307	390732341	0.996256787299
12	12382739	143693437	0.996192522703

图 14-19 预测运行结果

14.6　模型预测的准确性评测

在模型预测完成之后可以利用算法平台的评估组件对预测的准确性进行评测，评估组件与预测组件的输出端相连，右击该组件，选择“执行至此处”，即可完成评估，如图 14-20 所示。

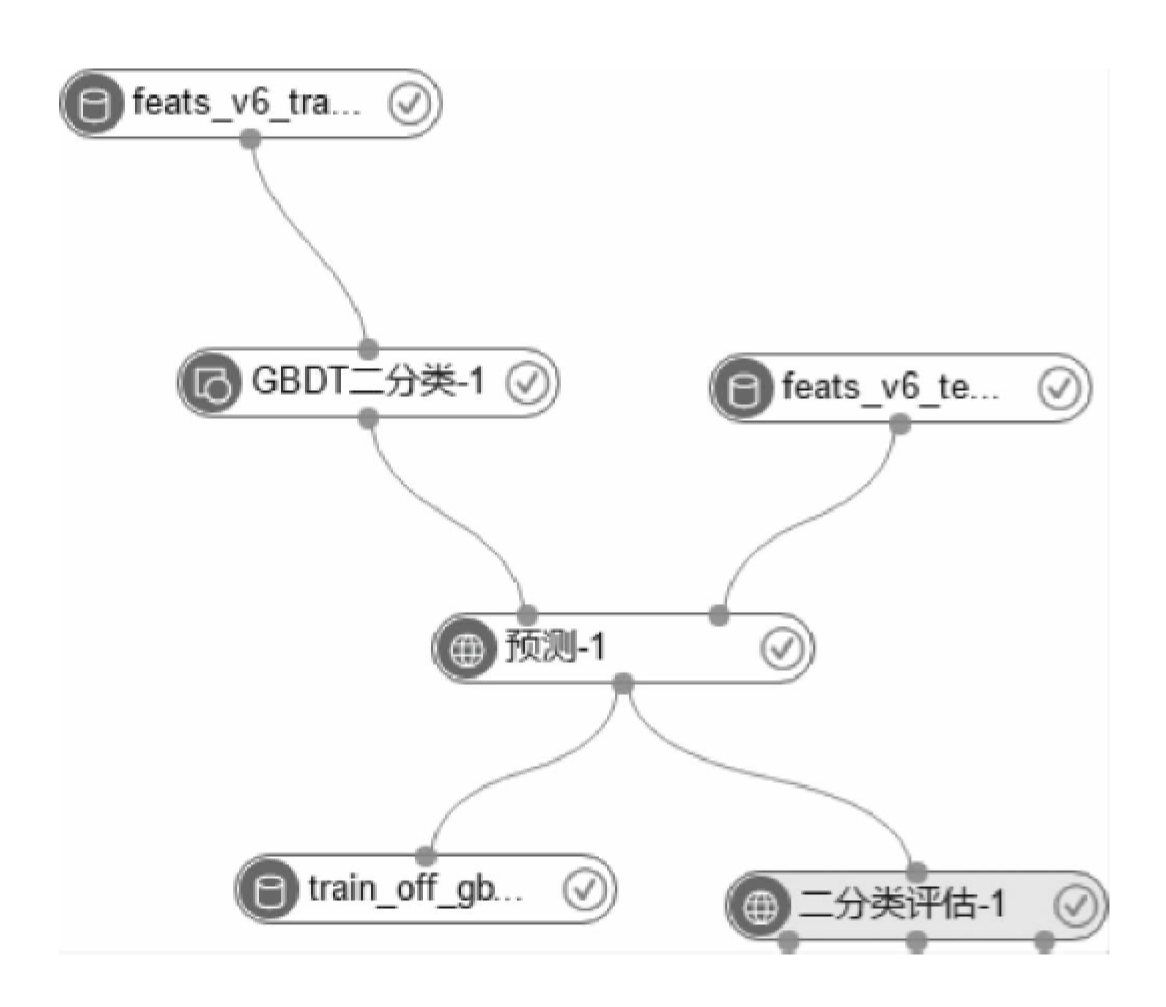
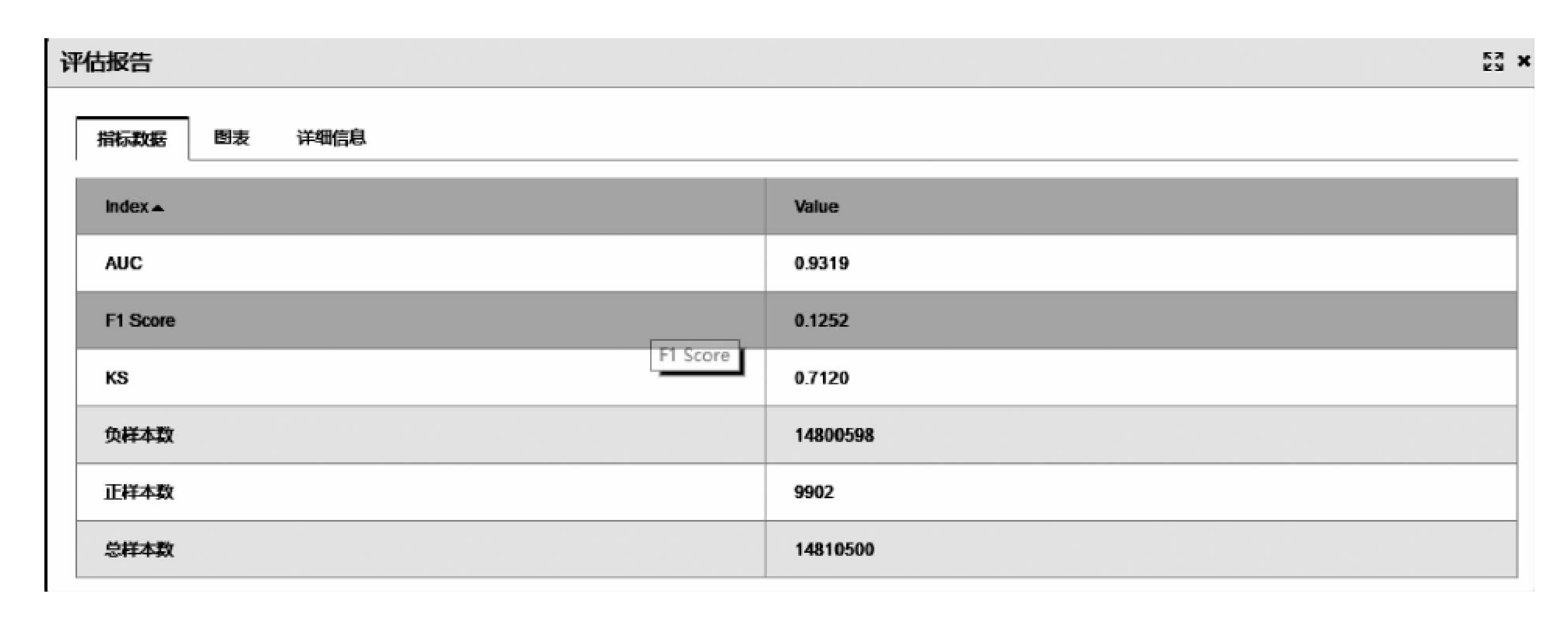


图 14-20　模型准确性预测评估

评估完成后右击该组件，选择“查看评估报告”，即可查看模型预测的效果，如图 14-21 所示。其中包括 AUC、F1 Score、KS 等评价指标，这些指标均用于评价二分类的分类效果，其取值范围均为[0,1]，值越大说明分类效果越好，在评估报告中还可以查看其对应的 ROC 曲线、PR 曲线、KS 曲线等图表，如图 14-22 所示。



评估报告

指标数据　图表　详细信息

Index	Value
AUC	0.9319
F1 Score	0.1252
KS	0.7120
负样本数	14800598
正样本数	9902
总样本数	14810500

图 14-21　准确性评估数据

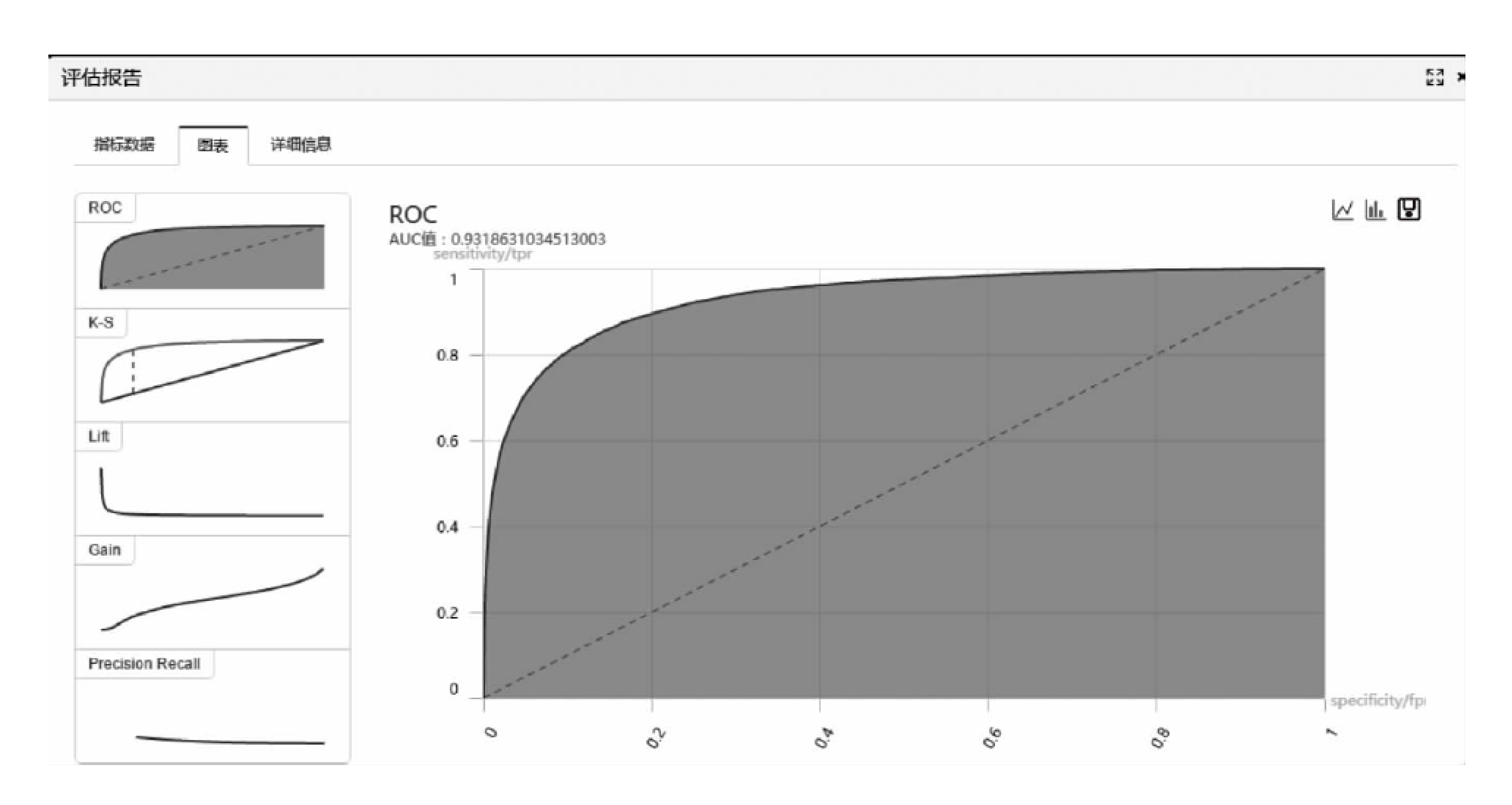


图 14-22 准确性评估图表

14.7 特征重要性的评估

为了提高预测的准确性，有时还需要观察特征的重要性，以便设计出更有效的特征，并去除无效的特征。

利用特征重要性评估组件可以方便地评估特征的重要性。图 14-23 展示了一个运用随机森林特征重要性评估组件来评估特征重要性的例子。评估完成后，以直方图的形式对特征重要性进行了展示，如图 14-24 所示。



图 14-23 随机森林特征重要性评估

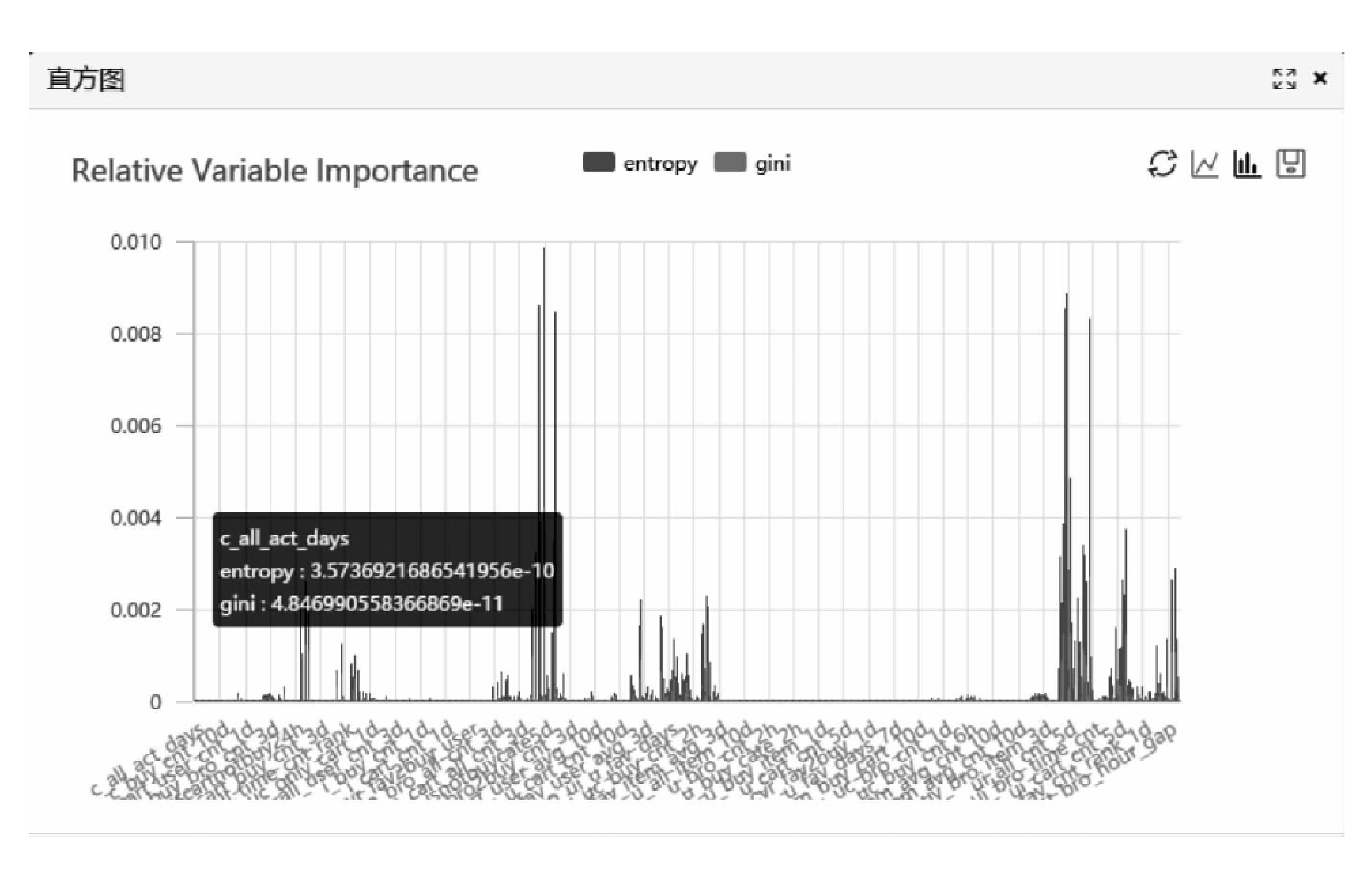


图 14-24　特征重要性评估结果

14.8　总结

本章展示了一个用数加平台实现的推荐系统，完整代码可以从 GitHub 上下载(https://github.com/alibook/alibook-bigdata.git)，该系统主要通过 MapReduce 程序提取用户与商品的特征，构造训练集，利用算法平台的 GBDT 组件训练了一个二分类模型来预测用户未来一天对于各商品发生购买行为的概率，并利用评估组件分别对预测的准确率和特征的重要性进行了评测，从中可以看出数加平台处理大数据的强大能力，同时该平台的界面也十分人性化，提供了非常丰富的组件，采用拖曳控件的方式即可完成模型的训练，非常易于上手。

参 考 文 献

［1］ 张俊林. 大数据日知录[M]. 北京：电子工业出版社，2014.
［2］ 杨巨龙. 大数据技术全解[M]. 北京：电子工业出版社，2014.
［3］ 黄宜华. 深入理解大数据[M]. 北京：机械工业出版社，2014.
［4］ 赵刚. 大数据技术与应用实践指南[M]. 北京：电子工业出版社，2013.
［5］ 李军. 大数据从海量到精准[M]. 北京：清华大学出版社，2014.
［6］ 陈工孟. 大数据导论[M]. 北京：清华大学出版社，2015.
［7］ 汤银才. R 语言与统计分析[M]. 北京：电子工业出版社，2012.

图书资源支持

感谢您一直以来对清华版图书的支持和爱护。为了配合本书的使用，本书提供配套的素材，有需求的用户请到清华大学出版社主页(http://www.tup.com.cn)上查询和下载，也可以拨打电话或发送电子邮件咨询。

如果您在使用本书的过程中遇到了什么问题，或者有相关图书出版计划，也请您发邮件告诉我们，以便我们更好地为您服务。

我们的联系方式：

地　　址：北京海淀区双清路学研大厦 A 座 707

邮　　编：100084

电　　话：010-62770175-4604

资源下载：http://www.tup.com.cn

电子邮件：weijj@tup.tsinghua.edu.cn

QQ：883604(请写明您的单位和姓名)

扫一扫

资源下载、样书申请

新书推荐、技术交流

用微信扫一扫右边的二维码，即可关注清华大学出版社公众号“书圈”。